SPAWAR
Systems Center
PACIFIC

TECHNICAL REPORT 2054
December 2014

Polar Codes

David Wasserman

SSC Pacific
San Diego, CA 92152-5001

SSC Pacific
San Diego, California 92152-5001

K. J. Rothenhaus, CAPT, USN
Commanding Officer

C. A. Keeney
Executive Director

ADMINISTRATIVE INFORMATION

The work described in this report was performed by the Space Systems Branch (Code 56270) of the ISR Division (Code 56200), Space and Naval Warfare Systems Center Pacific (SSC Pacific), San Diego, CA. The Naval Innovative Science and Engineering (NISE) Program at SSC Pacific funded this Applied Research project.

.

Released by
A. M. Mroczek, Head
Space Systems Branch

Under authority of
C. A. Wilgenbusch, Head
ISR Division

ACKNOWLEDGEMENT

The author would like to thank Justin James, Ahsen Ahmed, and Sergio Vargas, who did some of the work described in this report.

SB

EXECUTIVE SUMMARY

This report describes the results of the project “More reliable wireless communications through polar codes,” funded in fiscal year 2014 by the Naval Innovative Science and Engineering (NISE) program at SSC Pacific.

OBJECTIVE

The purpose of the project is to determine if polar codes can outperform the forward error correction currently used in Navy wireless communication systems. This report explains polar codes to non-specialists.

RESULTS

The project team has written and tested software that implements several published polar coding algorithms. For comparison, we have also implemented and tested other forward error correction methods: a turbo code, a low density parity check (LDPC) code, a Reed–Solomon code, and three convolutional codes.

CONTENTS

Figures

1. INTRODUCTION

Forward error correction (FEC) is a method for improving the reliability of digital communications. This is especially important for wireless communications, which are subject to noise, fading, and interference. It is common that a transmitter will send a 0, but the receiver receives a 1, or vice versa. FEC adds redundancy to digital messages before transmission. The goal of FEC is that even if some bits are flipped, the receiver can use the redundant information to reconstruct the original message.

The U.S. Navy uses FEC in most of its wireless communications. Navy systems use several types of FEC, with turbo codes being the most common. Many civilian systems use low density parity check (LDPC) FEC codes, and the Navy is planning to use LDPC for some future systems.

Polar codes are a new form of FEC. The first journal article about polar codes was [1] by E. Arikan, published in 2009. This paper made a great impact in the academic community. Polar codes are of interest primarily for theoretical reasons, but they may also have practical use as a replacement for turbo or LDPC codes.

The author of this report and his project team have studied polar codes to determine if they can improve the reliability and throughput of Navy wireless communications. This research was funded in fiscal year 2014 by the Naval Innovative Science and Engineering (NISE) program at SSC Pacific. We have found the polar coding literature hard to read. The main purpose of this report is to make it easier for others to learn what we have learned. In addition, many polar coding papers describe complex algorithms but do not provide source code. We have written our own source code to implement these algorithms [2], and this report should help guide the use of our code. Distribution of the source code is authorized only to U.S. Government agencies and their contractors.

The author has done his best to make this report readable by non-specialists. To understand this report, the reader should be comfortable with the basic concepts of probability, including conditional probability. The reader should also know a few common mathematical symbols for sets and functions.

2. BLOCK CODES

Let N and K be positive integers with $K \le N$. An (N, K) *block code* is a function from $\{0,1\}^K$ to $\{0,1\}^N$, i.e., its input is a vector of K bits and its output is a vector of N bits. An *encoder* is an implementation of the function in software or hardware.[1] N is called the *block size* or *block length*. K/N is called the *code rate*.

We usually use the symbol $\mathbf{u}$ for the code's input, and $\mathbf{x}$ for its output. The bits of $\mathbf{u}$ are called *information bits*, because they are the information that the sender wants the receiver to receive. The bits of $\mathbf{x}$ are called *code bits*. We send $\mathbf{x}$ through a channel, and the channel output is called $\mathbf{y}$. A *decoder* is a system designed to undo the effects of a particular encoder and channel. It takes $\mathbf{y}$ as input, and outputs $\hat{\mathbf{u}} \in \{0,1\}^K$, with the goal that $\hat{\mathbf{u}} = \mathbf{u}$. The *block error rate* (BLER) is the probability that $\hat{\mathbf{u}} \ne \mathbf{u}$, i.e., the probability that the block is not decoded correctly.[2] The *bit error rate* (BER) is the average probability that a bit is not decoded correctly. (We must say "average" because different positions with $\mathbf{u}$ may have different error probabilities.)

Example: Repetition code of rate 1/3. Let $K = 1$ and $N = 3$, and let the encoder map $0 \mapsto (0,0,0)$ and $1 \mapsto (1,1,1)$. Suppose the channel maps $\{0,1\}^3$ to $\{0,1\}^3$ so that each bit independently has a 10% chance of being flipped. Then the decoder should use the majority vote rule: if $\mathbf{y}$ is $(0,0,0)$, $(0,0,1)$, $(0,1,0)$, or $(1,0,0)$, then $\hat{\mathbf{u}} = 0$; if $\mathbf{y}$ is $(1,1,1)$, $(0,1,1)$, $(1,0,1)$, or $(0,1,1)$, then $\hat{\mathbf{u}} = 1$. The BLER and BER are both 0.028.

Example: (7, 4) Hamming code ([3]). Let $K = 4$ and $N = 7$. For any $u = (u_1, u_2, u_3, u_4) \in \{0,1\}^4$, the encoder maps u to

$$x = u \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

with the result computed modulo 2. This encoder maps the 16 vectors in $\{0,1\}^4$ to 16 different vectors in $\{0,1\}^7$ called *codewords*. These 16 codewords were chosen to satisfy the following property: each of the 128 vectors in $\{0,1\}^7$ is either a codeword or one bit different from a codeword. Suppose the channel maps $\{0,1\}^7$ to $\{0,1\}^7$ so that each bit independently has a 5% chance of being flipped. The decoder should find the codeword that is either equal to or one bit different from the received vector, and then compute the vector in $\{0,1\}^4$ that maps to that codeword. So the decoder will be correct if at most one bit gets flipped. Therefore the BLER is $1 - (0.95)^7 - 7(0.05)(0.95)^6 \approx 0.044$. The calculation of the BER is more complicated and will be omitted.

2.1 LATENCY

Latency, also called *delay*, is the time it takes for a particular information bit to travel from the sender to the receiver. The use of a block code adds to latency; we wish to determine how much. We make the following assumptions:

1. The information bits arrive at the encoder at a uniform rate ρ, which is called the *data rate* and is measured in bits per second (bps). The bits are stored in the encoder's input buffer until K of them have arrived. The encoder then unloads the buffer and processes the bits.

[1] Coding theorists often call this function an *encoder*, and define *code* as the image of the function. This report does not use these definitions.

[2] Some authors use the term *frame error rate*.

2. After the encoder has processed the bits for time T_e, the N output bits are available in the encoder's output buffer. They leave the output buffer at a uniform rate $\frac{N}{K}\rho$.

3. The decoder does the same in reverse: the N-bit input buffer is filled at rate $\frac{N}{K}\rho$, the decoder processes the block in time T_d, and the K-bit output buffer is emptied at rate ρ.

4. The encoder can only process one block at a time. It must be fast enough that when the last bit of block n arrives, it must be done encoding block $n-1$ so that it can immediately start encoding block n; otherwise some of the bits of block n will be overwritten by bits of block $n+1$. The same is true of the decoder. This assumption puts a limit on the data rate at which a particular encoder and decoder may be used.

The time to fill either input buffer or empty either output buffer is K/ρ. If block n begins arriving at the encoder at time t, then it finishes arriving at time $t + K/\rho$, and begins leaving at time $t + K/\rho + T_e$. So the latency added by the encoder is $K/\rho + T_e$. Similarly, the latency added by the decoder is $K/\rho + T_d$. By the last assumption, T_d and T_e are both $\leq K/\rho$, so the total latency added is between $2K/\rho$ and $4K/\rho$.

For example, suppose $K = 1024$, $T_e = 10^{-5}$ s, and $T_d = 10^{-4}$ s. If the data rate is 150 bps, then the coding adds 13.7 s of latency, which is unacceptable for many purposes. In contrast, if the data rate is 1 megabit per second (Mbps), then the coding adds 2.16 milliseconds (ms) of latency, which might be insignificant compared to other delays in the system. (For example, if the signal is relayed by a geosynchronous satellite, the propagation delay is about 250 ms.) The maximum possible data rate for this decoder is $K/T_d = 10.24$ Mbps.

3. INFORMATION THEORY

In [4], C.E. Shannon started the field of information theory, which is largely concerned with communication through channels. Figure 1 shows a simple model of a radio frequency (RF) communication link.

From the perspective of an RF engineer, the channel is the gap between the transmitter and the receiver, as shown in Figure 2. The input and output are both RF signals.

From the perspective of information theory, the channel is the gap between the encoder and the decoder, as shown in Figure 3. The input is a bit, and the output depends on the design of the demodulator.

An information theoretic *channel* is specified by a set $\mathcal{X}$ of input symbols, a set $\mathcal{Y}$ of output symbols, and for each $x \in \mathcal{X}$ and $y \in \mathcal{Y}$, a nonnegative real number called *transition probability* from x to y. If the channel is called W, then the transition probability from x to y is written as $W(y|x)$. $\mathcal{X}$ is usually $\{0, 1\}$, while $\mathcal{Y}$ may be discrete or continuous. If x is input to the channel, $W(y|x)$ is understood as the probability of the output being y. For every $x \in \mathcal{X}$, it is required that $\sum_{y\in\mathcal{Y}} W(y|x) = 1$ if $\mathcal{Y}$ is discrete, or that $\int_{\mathcal{Y}} W(y|x)\,dy = 1$ if $\mathcal{Y}$ is continuous.

Frequently, we know y and want to determine x. In this situation, $W(y|x)$ is also called the *likelihood* of x given y.

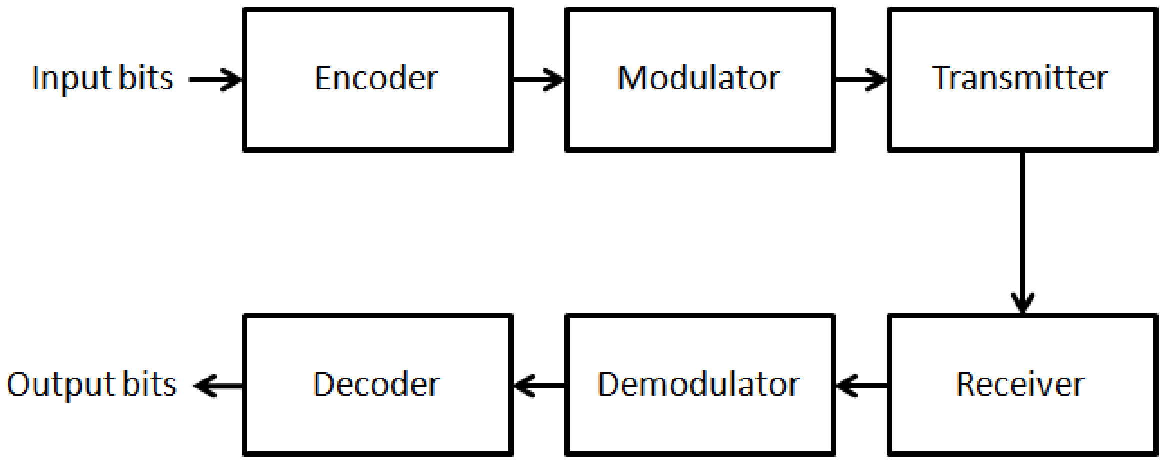

Figure 1. Simple RF link model.

Example 1: Gaussian Channel. Suppose we are using binary phase shift keying (BPSK) modulation and transmitting over a line-of-sight RF channel with negligible multipath. The Gaussian channel is used to model this situation. The BPSK modulator first maps a bit to a point in the in-phase/quadrature (I/Q) plane, $0 \mapsto (1, 0)$, $1 \mapsto (-1, 0)$. Henceforth, we ignore the Q component because it does not affect the demodulator's output. The I/Q point is then converted to a sinusoidal waveform which is transmitted. The receiver rescales the waveform to compensate for the attenuation. The demodulator maps the waveform back to I/Q space, and outputs the result. The overall effect is that $0 \mapsto 1 + n$ and $1 \mapsto -1 + n$, where n is a random noise variable that is assumed to be normally distributed with 0 mean.

For any $\sigma > 0$, the *Gaussian channel* with noise variance σ^2 has output set $\mathcal{Y} = \mathbb{R}$, and transition probabilities $W(y|0) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(y-1)^2}{2\sigma^2})$ (a normal distribution with mean 1 and standard deviation σ), $W(y|1) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(y+1)^2}{2\sigma^2})$ (a normal distribution with mean -1 and standard deviation σ). This channel is also called an *additive white Gaussian noise (AWGN)* channel.

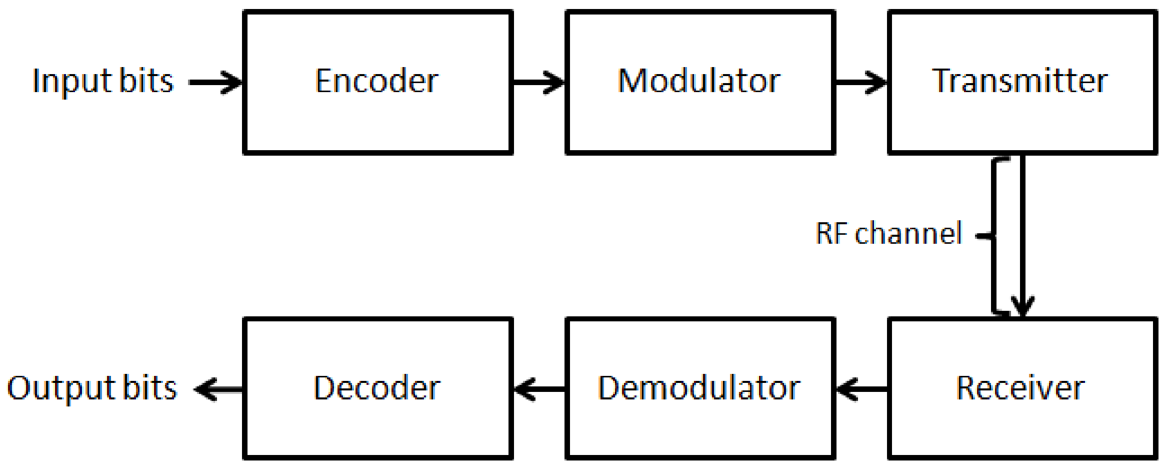

Figure 2. RF channel.

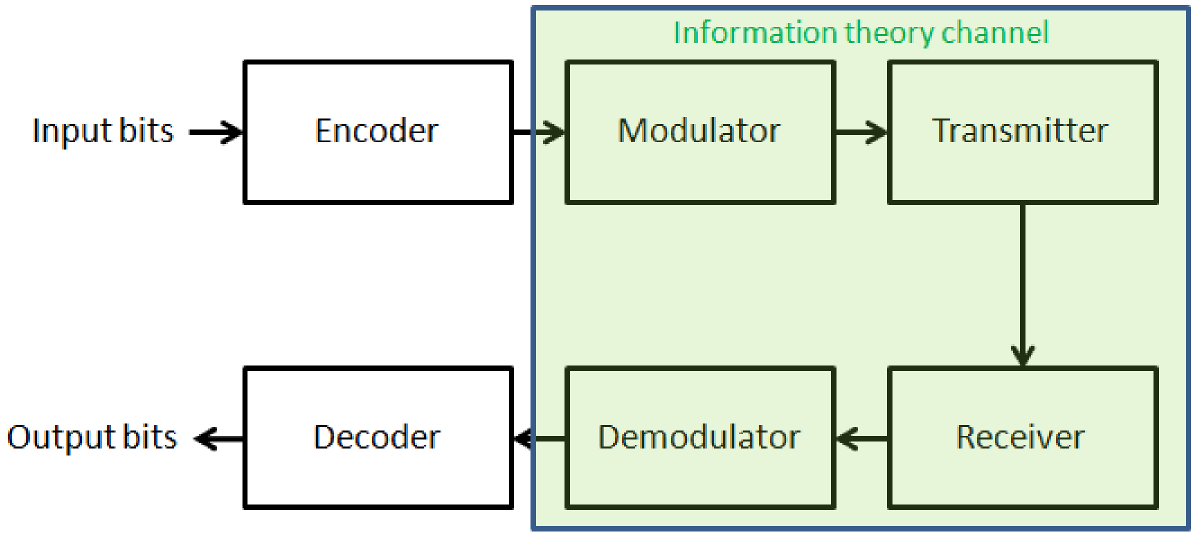

Figure 3. Information theory channel.

Example 2: Binary Symmetric Channel. In the previous example, the demodulator output a point in I/Q space. Suppose instead that the demodulator must decide if it has received a 0 or 1. It decides 0 if I is positive, and 1 if I is negative. The probability of error is $\int_{-\infty}^{0} \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(y-1)^2}{2\sigma^2})\, dy$.

For any $p \in [0,1]$, the *binary symmetric channel* (BSC) with error probability p has output set $\mathcal{Y} = \{0,1\}$, and transition probabilities $W(0|0) = W(1|1) = 1-p$, $W(0|1) = W(1|0) = p$.

The BSC is called a *hard decision* channel because it outputs a 0 or 1 and gives no other information. The AWGN is called a *soft decision* channel because it indicates the degree of uncertainty in estimating the channel input. This provides more information for the decoder to estimate the encoder input.

All channels considered in this report will be binary memoryless symmetric (BMS) channels unless stated otherwise. *Binary* means that the input set is $\mathcal{X} = \{0,1\}$. *Memoryless* means that the when the channel is used multiple times, the transition probabilities of each use are independent of the inputs and outputs of other uses. A channel W is *symmetric* if there is a one-to-one correspondence $c : \mathcal{Y} \to \mathcal{Y}$ such that for all $y \in \mathcal{Y}$,

1. $c(c(y)) = y$;
2. $W(y|0) = W(c(y)|1)$;
3. $W(y|1) = W(c(y)|0)$.

$c(y)$ is usually written $\bar{y}$. For the BSC, $\bar{0} = 1$ and $\bar{1} = 0$. For the AWGN channel, $\bar{y} = -y$.

3.1 CHANNEL CAPACITY

Shannon showed how to compute the *capacity* $C(W)$ of a channel W, which is the highest rate at which information can be sent through the channel with arbitrarily low BER. Formally, he proved that for any rational number $r < C(W)$, and every $\epsilon > 0$, there is an code with rate r and a corresponding decoder that can decode the channel output with BER $< \epsilon$, but this is not possible for any $r > C(W)$. Shannon's proof did not show how to explicitly construct such a code. Furthermore, the decoder in his proof is a *maximum likelihood* decoder. For any $\mathbf{u} \in \{0,1\}^K$, the *likelihood* of $\mathbf{u}$ is the probability that this $\mathbf{u}$ would produce the given $\mathbf{y}$. The maximum likelihood decoder computes the likelihood of all 2^K possible $\mathbf{u}$'s, and outputs the one with the maximum likelihood. This requires $O(2^K)$ steps, which is not practical except for very small values of K.

Figure 4 shows the capacity of a BSC as a function of the error probability p. Figure 5 shows the capacity of a AWGN channel as a function of the noise standard deviation σ.

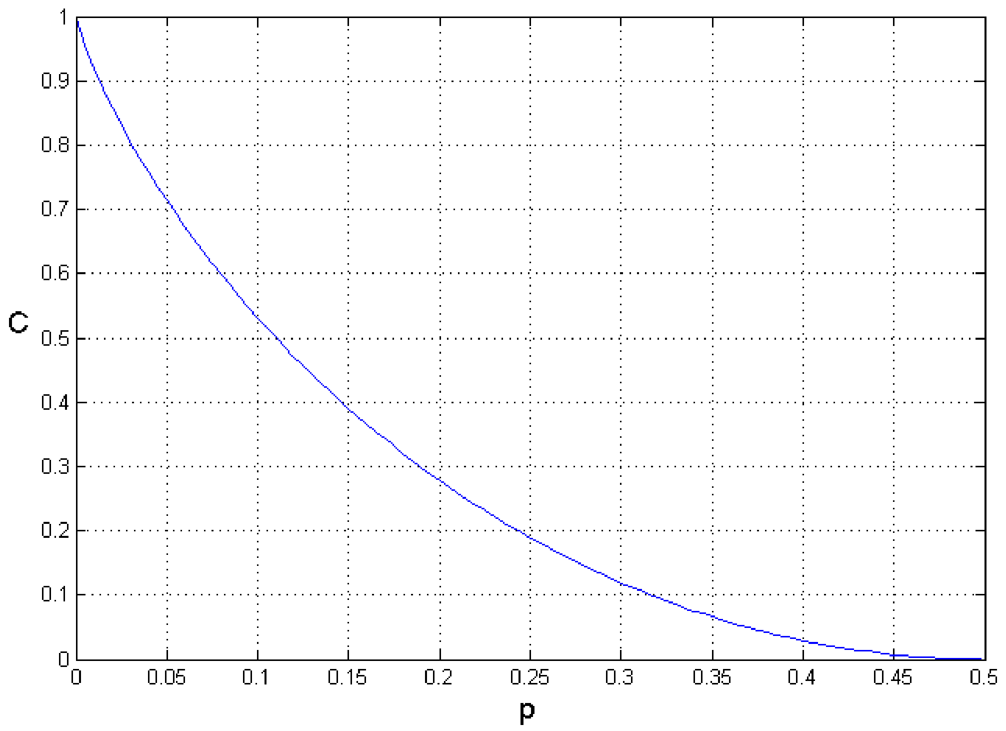

Figure 4. Capacity of BSC.

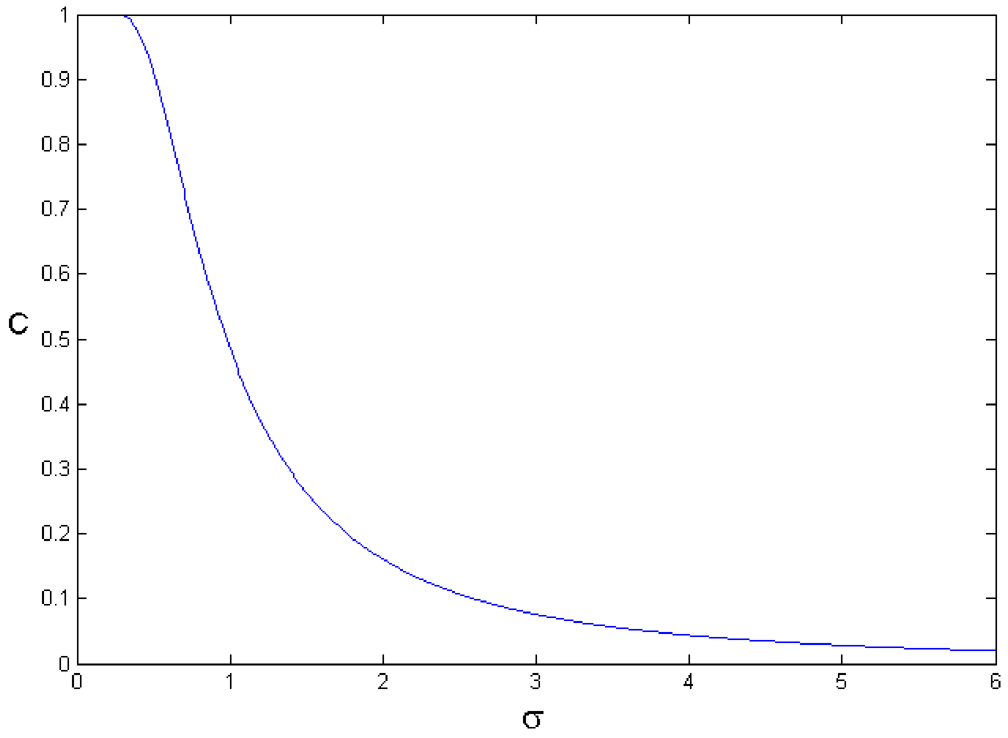

Figure 5. Capacity of AWGN channel.

4. INTRODUCTION TO POLAR CODES

Polar codes were introduced by E. Arikan in [1]. This paper showed how to construct polar encoders and decoders for any block length N that is a power of 2, and any $K \leq N$. These encoders and decoders are relatively efficient, requiring $O(N \log N)$ instructions, as opposed to $O(2^K)$ for a maximum likelihood decoder. They were the first efficient encoders and decoders proven to *achieve the capacity* of any BMS channel. This is not as useful as it sounds. Achieving the capacity of a channel W does not mean exhibiting a code with rate $C(W)$. It means exhibiting an infinite sequence of codes such that as N goes to infinity, the code rate converges to $C(W)$ and BER converges to 0. So to produce a polar code with a rate very close to the channel capacity, we may need a block length that is too large to use.

Reference [5] states that polar codes by themselves have underperformed state-of-the-art codes of similar block lengths and code rates. This may not be the most relevant comparison because the efficiency of polar codes may make them feasible at larger block lengths than other codes; on the other hand, larger block lengths may cause unacceptable latency.

Several authors have achieved better results by combining polar codes with other codes, for example, [5–7]. In particular, I. Tal and A. Vardy [5] combined a polar code with a cyclic redundancy check (CRC) and found that the resulting code of length 2048 outperformed the length 2304 LDPC code used in the WiMax standard. In [8, 9], K. Nui, K. Chen, and J.R. Lin combined polar codes of length 512, 1024, and 2048 with a CRC. They also combined this CRC with turbo codes of the same lengths, and found that the polar codes outperformed the turbo codes. However, [8, 9] compared BLERs and did not give any BER results.

Polar codes can also be used for source coding (i.e., compression), but this use will not be discussed in this report.

5. POLAR ENCODER

An (N, K) polar code is a block code with K inputs and N outputs. However, we will begin by introducing a function G_N that has N bit inputs, which are numbered from 1 to N, and N bit outputs, also numbered from 1 to N. The input to G_N is a row vector called $\mathbf{u} = (u_1, u_2, \ldots, u_N)$, where u_i is the bit that goes into input i.[3] This disagrees with the notation in Section 2, where we stated that $\mathbf{u}$ is a vector of length K. The output of G_N is a row vector of length N called $\mathbf{x} = (x_1, x_2, \ldots, x_N)$.

To construct a polar encoder, we choose K of G_N's inputs, and put our information bits into these inputs, while the remaining $N - K$ inputs are *frozen*, i.e., held constant. The frozen bits are normally set to 0, but they may have any value that is known to both the encoder and the decoder. Figure 6 shows an $(8, 4)$ polar encoder in which the frozen bits are u_1, u_2, u_3, and u_5. The set $\{4, 6, 7, 8\}$ is denoted $\mathcal{A}$ and is called the *information set*, because the information bits are sent to inputs 4, 6, 7, and 8 of G_8. The complement of $\mathcal{A}$ is $\mathcal{A}^C = \{1, 2, 3, 5\}$, which is called the *frozen set* because inputs 1, 2, 3, and 5 of G_8 are frozen. The input to the encoder is called $\mathbf{u}(\mathcal{A})$. The output from the encoder is the same as the output from G_N, and is called $\mathbf{x}$.

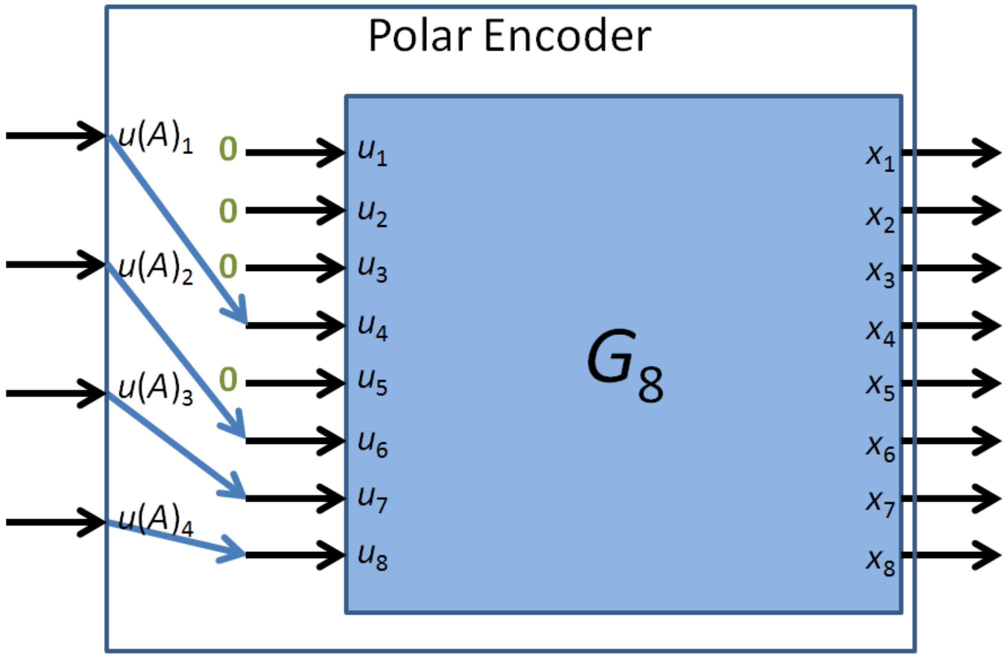

Figure 6. (8, 4) polar encoder.

This encoder is implemented in the MATLAB® file polarEncode.m.

5.1 CHOOSING $\mathcal{A}$

Let n be any positive integer and let $N = 2^n$. For any $K \leq N$, we can construct an (N, K) polar code by choosing $\mathcal{A}$ to be any K-element subset of $\{1, 2, \ldots, N\}$. However, we must choose $\mathcal{A}$ carefully to get a good code. The method for choosing $\mathcal{A}$ is that we imagine decoding all N inputs of G_N with no frozen bits, and determine the probability of decoding error for each input. These probabilities depend on

[3]For some purposes it is more convenient to used indices 0 through $N - 1$, and many polar coding papers use this convention. We start at 1 to agree with MATLAB® indexing.

the channel W. We optimize the polar code for W by choosing $\mathcal{A}$ as the set of inputs with the lowest error probabilities.

5.2 STRUCTURE OF G_N

G_N is defined by a simple recursive rule. Figures 7 and 8 show diagrams of G_2, G_4, and G_8. In these figures, $\oplus$ represents the modulo two sum, or equivalently, the exclusive or operator.

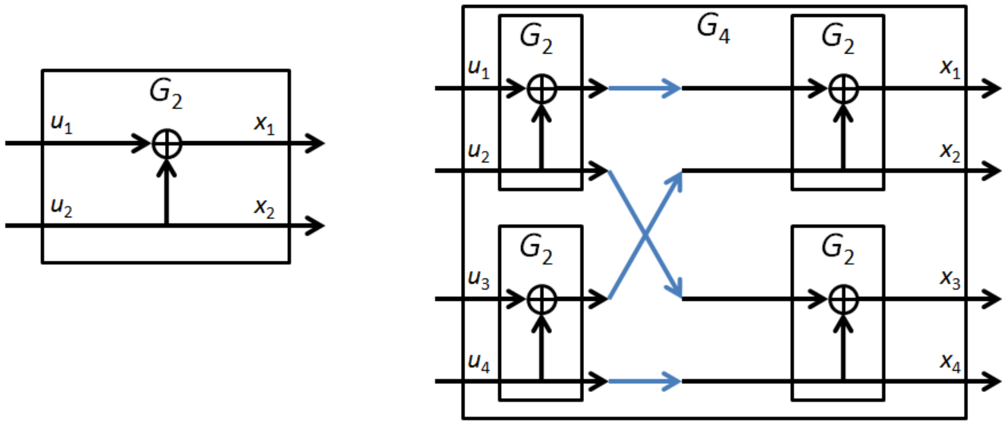

Figure 7. G_2 and G_4.

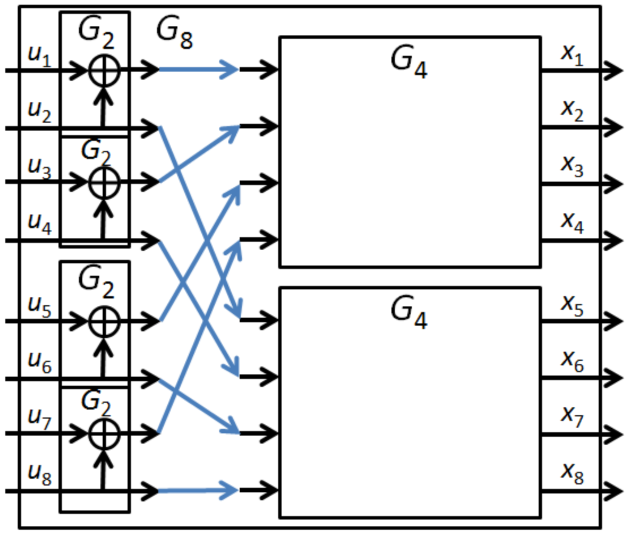

Figure 8. G_8.

In general, G_{2N} is constructed using N copies of G_2 and two copies of G_N, as shown in Figure 9. The box marked R_{2N} is a permutation called the *reverse shuffle*, which copies its odd-numbered inputs to its first N outputs, and copies its even-numbered inputs to its last N outputs. So the recursive formula for $G_{2N}(u_1, \ldots, u_{2N})$ is $(G_N(u_1 \oplus u_2, u_3 \oplus u_4, \ldots, u_{2N-1} \oplus u_{2N}), G_N(u_2, u_4, \ldots, u_{2N}))$.

G_N is implemented in the MATLAB® file W_N.m.

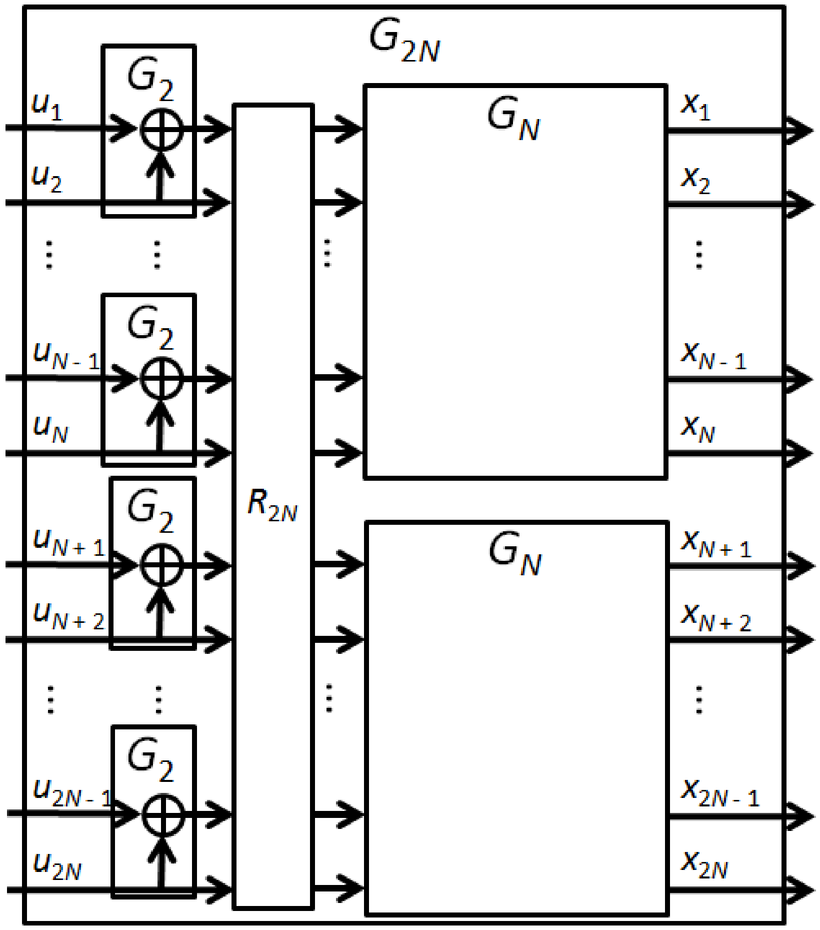

Figure 9. Recursive construction of G_{2N}.

6. POLAR DECODER

The decoder described in [1] is called a *successive cancellation* (SC) decoder. It makes bit decisions one at a time. When decoding the ith bit, the data it uses are the channel output $\mathbf{y}$, and the previous bit decisions $\hat{u}_1$ through $\hat{u}_{i-1}$, which it assumes are correct. It uses the following rules:

- If the ith input is frozen, then u_i is known, so set $\hat{u}_i = u_i$.
- If the ith input is not frozen, then choose $\hat{u}_i$ to be the bit value that would have a higher probability to produce the channel output $\mathbf{y}$. This probability is denoted $W_N^{(i)}(y_1, \ldots, y_N, \hat{u}_1, \ldots, \hat{u}_{i-1}|u_i)$.

To compute $W_N^{(i)}(y_1, \ldots, y_N, \hat{u}_1, \ldots, \hat{u}_{i-1}|u_i = 0)$, choose bit values u_{i+1} through u_N, and let

$$\mathbf{x} = G_N(\hat{u}_1, \ldots, \hat{u}_{i-1}, 0, u_{i+1}, \ldots, u_N),$$

i.e., the encoder output when $(\hat{u}_1, \ldots, \hat{u}_{i-1}, 0, u_{i+1}, \ldots, u_N)$ is the input to G_N. The probability of sending $\mathbf{x}$ through the channel and receiving $\mathbf{y}$ is $\prod_{i=1}^{N} W(y_i|x_i)$. We average this value over all 2^{N-i} choices of u_{i+1} through u_N, and the result is $W_N^{(i)}(y_1, \ldots, y_N, \hat{u}_1, \ldots, \hat{u}_{i-1}|u_i = 0)$.

We then compute $W_N^{(i)}(y_1, \ldots, y_N, \hat{u}_1, \ldots, \hat{u}_{i-1}|u_i = 1)$ the same way. If

$$W_N^{(i)}(y_1, \ldots, y_N, \hat{u}_1, \ldots, \hat{u}_{i-1}|u_i = 0) \geq W_N^{(i)}(y_1, \ldots, y_N, \hat{u}_1, \ldots, \hat{u}_{i-1}|u_i = 1), \tag{1}$$

we choose $\hat{u}_i = 0$; otherwise, we choose $\hat{u}_i = 1$. [4]

If we applied this rule directly, the complexity of the decoder would be $\Theta(2^N)$. Fortunately, [1] describes a recursive method that reduces the complexity to $\Theta(N \log N)$.

After the decoder is done computing $\hat{u}_1$ through $\hat{u}_N$, it outputs $\hat{u}(A)$, the estimates of the K non-frozen inputs.

6.1 LENGTH 2 DECODER

We now describe the decoder in more detail for the case $N = 2$. Figure 10 shows how the decoder's input is formed. The encoder maps the u's (which may include frozen bits) to the x's, which are transmitted through the channel W, and the y's are received. The y's may be bits or soft decisions. It is assumed that we know the complete description of W.

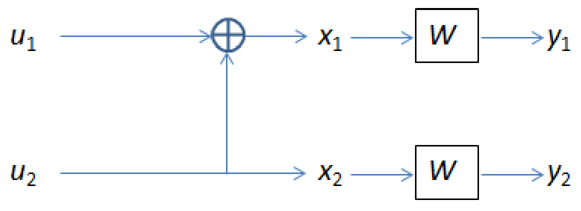

Figure 10. Length 2 encoding and transmission.

Observe $x_1 = u_1 \oplus u_2$ and $x_2 = u_2$. Also $u_1 = x_1 \oplus x_2$ and $u_2 = x_2$, so G_2 is its own inverse.

[4] Observe that ties are broken by choosing 0. This is the rule from [1]. For some purposes it is better to break ties randomly.

For any y received, we know the transition probabilities $W(y|x = 0)$ and $W(y|x = 1)$. Let $a = W(y_1|x_1 = 0)$, $b = W(y_1|x_1 = 1)$, $c = W(y_2|x_2 = 0)$, $d = W(y_2|x_2 = 1)$. These four numbers are all the data needed by the decoder; the values of y_1 and y_2 are not needed.

To decode u_1, we compute and compare the probabilities $W(y_1, y_2|u_1 = 0)$ and $W(y_1, y_2|u_1 = 1)$. The y's also depend on u_2, which is equally likely to be 0 or 1. So

$$\begin{aligned} W(y_1, y_2|u_1 = 0) &= W(y_1, y_2|u_1 = 0, u_2 = 0)/2 + W(y_1, y_2|u_1 = 0, u_2 = 1)/2 \\ &= W(y_1, y_2|x_1 = 0, x_2 = 0)/2 + W(y_1, y_2|x_1 = 1, x_2 = 1)/2. \end{aligned}$$

Since the transitions $x_1 \to y_1$ and $x_2 \to y_2$ are independent, we can factor the pairwise probabilities:

$$\begin{aligned} W(y_1, y_2|u_1 = 0) &= W(y_1|x_1 = 0)W(y_2|x_2 = 0)/2 + W(y_1|x_1 = 1)W(y_2|x_2 = 1)/2 \\ &= (ac + bd)/2. \end{aligned}$$

Similarly,

$$\begin{aligned} W(y_1, y_2|u_1 = 1) &= W(y_1, y_2|u_1 = 1, u_2 = 0)/2 + W(y_1, y_2|u_1 = 1, u_2 = 1)/2 \\ &= W(y_1, y_2|x_1 = 1, x_2 = 0)/2 + W(y_1, y_2|x_1 = 0, x_2 = 1)/2 \\ &= W(y_1|x_1 = 1)W(y_2|x_2 = 0)/2 + W(y_1|x_1 = 0)W(y_2|x_2 = 1)/2 \\ &= (bc + ad)/2. \end{aligned}$$

The ratio of these probabilities is $(ac + bd)/(bc + ad)$. Dividing both numerator and denominator by bd, we get

$$\frac{\frac{a}{b}\frac{c}{d} + 1}{\frac{c}{d} + \frac{a}{b}}.$$

This is the *likelihood ratio* (LR) of u_1 given y_1 and y_2, and it can be expressed in terms of a/b and c/d, the LRs of x_1 and x_2. We define the function $f(p, q) = \frac{pq+1}{p+q}$ [10].

We choose $\hat{u}_1 = 0$ if $f(a/b, c/d) \geq 1$; otherwise, we choose $\hat{u}_1 = 1$.

The next step is to decode u_2, so we determine which value of u_2 would have a higher probability of producing the observed y_1 and y_2. We compute the LR $W(y_1, y_2|u_2 = 0)/W(y_1, y_2|u_2 = 1)$. According to the successive cancellation (SC) rule, we assume that we have correctly decoded u_1. If $u_1 = 0$, then our LR becomes

$$\begin{aligned} \frac{W(y_1, y_2|u_1 = 0, u_2 = 0)}{W(y_1, y_2|u_1 = 0, u_2 = 1)} &= \frac{W(y_1, y_2|x_1 = 0, x_2 = 0)}{W(y_1, y_2|x_1 = 1, x_2 = 1)} \\ &= ac/bd \\ &= (a/b)(c/d). \end{aligned}$$

If $u_1 = 1$, then our LR becomes

$$\begin{aligned} \frac{W(y_1, y_2|u_1 = 1, u_2 = 0)}{W(y_1, y_2|u_1 = 1, u_2 = 1)} &= \frac{W(y_1, y_2|x_1 = 1, x_2 = 0)}{W(y_1, y_2|x_1 = 0, x_2 = 1)} \\ &= bc/ad \\ &= (a/b)^{-1}(c/d). \end{aligned}$$

In either case, we again see that the LR of u_2 can be expressed in terms of a/b and c/d, the LRs of x_1 and x_2. We define the function $g(p, q, u_1) = p^{1-2u_1}q$ [10].

We choose $\hat{u}_2 = 0$ if $g(a/b, c/d, \hat{u}_1) \geq 1$; otherwise we choose $\hat{u}_2 = 1$.

Therefore the decoder does not need all four values a, b, c, and d. Having LRs a/b and c/d is sufficient.

One may ask, if y_2 is a soft estimate of x_2, and $x_2 = u_2$, why don't we use the same estimate for u_2? If x_2 is more likely to be 0 than 1 (or vice versa), then isn't the same true for u_2? Usually it is. An exception can occur when the LR of u_1 is < 1 (indicating u_1 is more likely 1) but u_1 is a frozen 0. This means that a bit got flipped in transmission. If y_1 is a strong 0 and y_2 is a weak 1, then y_2 is more likely to be incorrect, so we conclude $x_2 = u_2 = 0$.

In the next section, estimates of x_1 and x_2 will also be necessary. These are computed $\hat{x}_1 = \hat{u}_1 \oplus \hat{u}_2$ and $\hat{x}_2 = \hat{u}_2$.

6.2 EFFICIENT LENGTH N DECODER

The decoder needs the following inputs:

- The LRs of the N encoder outputs
- The set $A \subseteq \{1, \ldots, N\}$ of frozen indices
- $u(\mathcal{A}^C)$, the known values of the frozen bits

In Figure 7, G_4 has three columns that each contain four bit values: the input column, the output column, and the middle column that has the bits $u_1 \oplus u_2$, u_2, $u_3 \oplus u_4$, and u_4. G_4 also contains four copies of G_2: two on the left that transform the input bits to the middle bits, and two on the right that transform the middle bits to the output bits. Similarly, in Figure 8, we can mentally fill in the missing details of the G_4's, and see that G_8 has four columns that each contain eight bit values, and between these columns are three columns that each contain four copies of G_2. In general, for $N = 2^n$, G_N has $n + 1$ columns that each contain N bit values, and n columns that each contain $N/2$ copies of G_2.

The decoder works by following the structure of G_N, repeating the procedure in Section 6.1 for each of the $nN/2$ copies of G_2. In this way, it retraces the encoder's steps, and produces a LR and a bit estimate for each of the $(n + 1)N$ bits in the encoder. The decoder has $nN/2$ elements that each represent one copy of G_2. The connections between these elements parallel the connections in the decoder. LRs are input on the right side of the decoder (representing the encoder output) and are computed from right to left. Bit estimates are first made at the left, and are computed from left to right.

For any bit b in the encoder, let $L(b)$ be its LR, and let $\hat{b}$ be the decoder's estimate. Each decoding element follows the steps listed below. In these steps we will use the symbols u_1, u_2, x_1, and x_2 for the inputs and outputs of a particular copy of G_2, regardless of its place in G_N. One element's x_1 may be another element's u_1 or u_2.

1. If this element is in the rightmost column, then $L(x_1)$ and $L(x_2)$ are included in the decoder's input. Otherwise, wait until they are computed by elements in the next column to the right.
2. Set $L(u_1) = f(L(x_1), L(x_2))$.
3. If this element is in the leftmost column, then set $\hat{u}_1$ using the decision rule: if this is a frozen bit, use the known value, otherwise set $\hat{u}_1 = 0$ if $L(u_1) \geq 1$, or $\hat{u}_1 = 1$ if $L(u_1) < 1$. If this element is not in the leftmost column, wait for $\hat{u}_1$ to be computed by an element in the next column to the left.

4. Set $L(u_2) = g(L(x_1), L(x_2), \hat{u}_1)$.

5. If this element is in the leftmost column, then set $\hat{u}_2$ using the decision rule: if this is a frozen bit, use the known value, otherwise set $\hat{u}_2 = 0$ if $L(u_2) \geq 1$, or $\hat{u}_2 = 1$ if $L(u_2) < 1$. If this element is not in the leftmost column, wait for $\hat{u}_2$ to be computed by an element in the next column to the left.

6. Set $\hat{x}_1 = \hat{u}_1 \oplus \hat{u}_2$ and $\hat{x}_2 = \hat{u}_2$.

We now return to using u_i and x_i to represent the inputs and outputs of G_N. This decoding method has complexity $O(N \log N)$ because there are $N \log N$ decoding elements, each of which executes steps 1 through 6 once. Reference [1] proves that this method produces the same results as direct application of the SC rule, Equation (1). At the time that $\hat{u}_i$ is decided, $\hat{u}_1$ through $\hat{u}_{i-1}$ have already been decided, and

$$L(u_i) = \frac{W_N^{(i)}(y_1, \ldots, y_N, \hat{u}_1, \ldots, \hat{u}_{i-1} | u_i = 0)}{W_N^{(i)}(y_1, \ldots, y_N, \hat{u}_1, \ldots, \hat{u}_{i-1} | u_i = 1)}.$$

6.3 RECURSIVE IMPLEMENTATION OF THE DECODER

Recall that Figure 9 shows how to construct G_N using two copies of $G_{N/2}$ and $N/2$ copies of G_2. This suggests that the decoder could be implemented with a recursive function. This function would have to output the LRs $L(u_1)$ through $L(u_N)$ in addition to the bit decisions $\hat{u}_1$ through $\hat{u}_N$. It would perform the following steps:

1. Call itself to decode $(x_1, \ldots, x_{N/2})$ with no frozen bits. Let the output LRs be called $L(v_1)$ through $L(v_{N/2})$. The output bit decisions are not used.

2. Call itself to decode $(x_{N/2+1}, \ldots, x_N)$ with no frozen bits. Let the output LRs be called $L(v_{N/2+1})$ through $L(v_N)$. The output bit decisions are not used.

3. Use the method of Section 6.1 $N/2$ times, using the appropriate $L(v)$'s as input, to compute $L(u_1)$ through $L(u_N)$ and $\hat{u}_1$ through $\hat{u}_N$.

Such an implementation would be incorrect, because in steps 1 and 2 this function would decide the $\hat{v}$'s using the decision rule, but the $\hat{v}$'s should be computed from the $\hat{u}$'s. Although the $\hat{v}$'s are not needed by the calling function, they are needed internally. For example, $\hat{v}_1$ is used to compute $L(v_2)$.

Fortunately, we will show that G_N has another decomposition that leads to a recursive decoder function.

The following result may be well-known, but we have not seen it explicitly stated in published literature:

Theorem 1. *For any $N = 2^n$, G_N is its own inverse, i.e., for any $\mathbf{u} \in \{0,1\}^N$, $G_N(G_N(\mathbf{u})) = \mathbf{u}$.*

Proof. Let $\mathbb{F}_2$ be the field with two elements 0 and 1. G_N is a linear map from $\mathbb{F}_2^N$ to $\mathbb{F}_2^N$, so it can be described as multiplication by an N by N matrix, which will also be called G_N. Equation (70) of [1] shows that $G_N = B_N F^{\otimes n}$, where $F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, $F^{\otimes n}$ is its nth tensor power, and B_N is a square matrix called the *bit-reversal operator*. Therefore $G_N^{-1} = (F^{\otimes n})^{-1} B_N^{-1}$. Section VII.B of [1] shows that $B_N^{-1} = B_N$.

Also we see by direct computation that $FF = I_2$. Using the tensor product identity $(AC) \otimes (BD) = (A \otimes B)(C \otimes D)$, we get that $(F \otimes F)(F \otimes F) = I_2 \otimes I_2 = I_4$. By induction on n, we get $F^{\otimes n} F^{\otimes n} = I_N$, so $(F^{\otimes n})^{-1} = F^{\otimes n}$. Combining these equations, we get $G_N^{-1} = F^{\otimes n} B_N$. Finally, Proposition 16 of [1] shows that $F^{\otimes n} B_N = G_N$. □

We can make a diagram of G_N^{-1} by flipping Figure 9 horizontally, thus exchanging the roles of the inputs and outputs. Since $G_N^{-1} = G_N$, this flipped diagram provides a second decomposition of G_N, as shown in Figure 11.

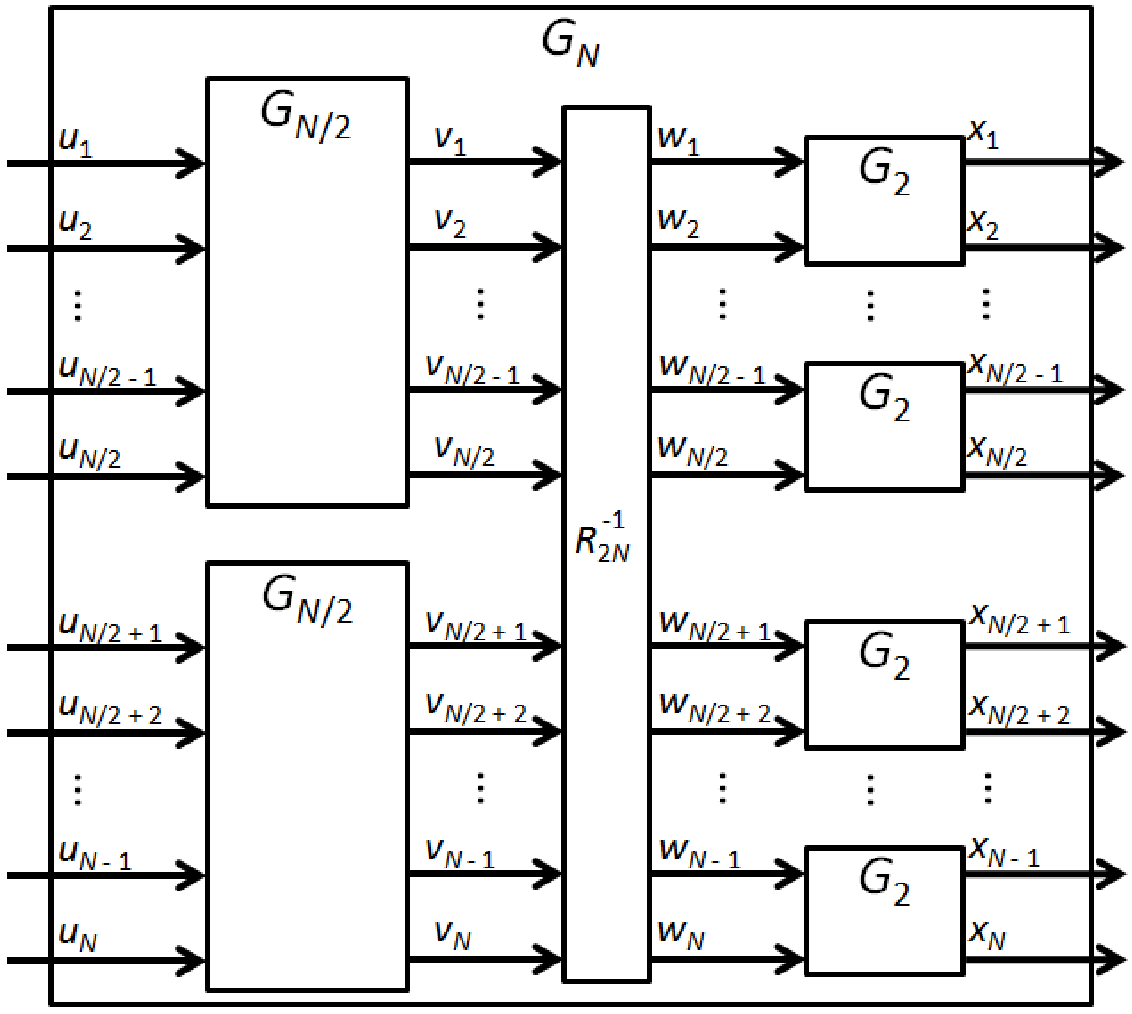

Figure 11. Second decomposition of G_N.

(We have also replaced $2N$ with N, i.e., this figure shows G_N in terms of $G_{N/2}$, instead of G_{2N} in terms of G_N.) When R_N is flipped horizontally, it becomes R_N^{-1}, which could be called the “shuffle”:

$$R_N^{-1}(v_1, \ldots, v_N) = (v_1, v_{N/2+1}, v_2, v_{N/2+2}, \ldots, v_{N/2}, v_N).$$

$G_{N/2}$ and G_2 also become their inverses, but by Theorem 1 this does not make a difference.

Now we can describe how to implement the decoder with a recursive function, using an algorithm based on [11]. Recall that the inputs to the decoder are $L(x_1)$ through $L(x_N)$, the set $\mathcal{A}$ of non-frozen indices, and $u(\mathcal{A}^C)$, the frozen bit values. The recursive function must output both $\hat{\mathbf{u}}$ and $\hat{\mathbf{x}}$. The decoder executes these steps:

1. Split $\mathcal{A}$ into two pieces $\mathcal{A}_1 = \mathcal{A} \cap \{1, \ldots, N/2\}$ and $\mathcal{A}_2 = \mathcal{A} \cap \{N/2+1, \ldots, N\}$, These are the sets of non-frozen indices needed for the two recursive calls. Subtract $N/2$ from each element of $\mathcal{A}_2$, because $u_{N/2+1}$ through u_N are called u_1 through $u_{N/2}$ within the second recursive call.
2. Split $u(\mathcal{A}^C)$ into $u(\mathcal{A}_1^C)$ and $u(\mathcal{A}_2^C)$ in a similar way.
3. For $1 \leq i \leq N/2$, compute $L(w_{2i-1}) = f(L(x_{2i-1}), L(x_{2i}))$. This is step 2 from Section 6.2, repeated for each of the $N/2$ copies of G_2 shown in Figure 11.
4. Call itself with inputs $(L(w_1), L(w_3), \ldots, L(w_{N-1}))$, $\mathcal{A}_1$, and $u(\mathcal{A}_1^C)$, producing outputs $(\hat{u}_1, \ldots, \hat{u}_{N/2})$ and $(\hat{v}_1, \ldots, \hat{v}_{N/2})$.
5. For $1 \leq i \leq N/2$, compute $L(w_{2i}) = g(L(x_{2i-1}), L(x_{2i}), \hat{v}_i)$ (step 4 of Section 6.2).
6. Call itself with inputs $(L(w_2), L(w_4), \ldots, L(w_N))$, $\mathcal{A}_2$, and $u(\mathcal{A}_2^C)$, producing outputs $(\hat{u}_{N/2+1}, \ldots, \hat{u}_N)$ and $(\hat{v}_{N/2+1}, \ldots, \hat{v}_N)$.
7. For $1 \leq i \leq N/2$, compute $\hat{x}_{2i-1} = \hat{v}_i \oplus \hat{v}_{N/2+i}$ and $\hat{x}_{2i} = \hat{v}_{N/2+i}$ (step 6 of Section 6.2).
8. Return $\hat{\mathbf{u}}$ and $\hat{\mathbf{x}}$.

This procedure has one minor shortcoming; it returns all N elements of $\hat{\mathbf{u}}$, instead of $\hat{\mathbf{u}}(\mathcal{A})$, which is the estimate of the length K encoder input. In our MATLAB® implementation, polarDecode.m version 3, this problem is handled by using a helper function to execute the above steps. The `polarDecode` function calls the helper function to obtain $\hat{\mathbf{u}}$ (ignoring the $\hat{\mathbf{x}}$ output) and then returns $\hat{\mathbf{u}}(\mathcal{A})$.

6.4 NUMERICAL IMPLEMENTATION

Recall that $f(x, y) = \frac{xy+1}{x+y}$ is one of the functions we use to compute LRs. One difficulty in using this function is that when we use it repeatedly, the LRs get closer to 1. The approach can be quite rapid: if δ and ϵ are small, then $f(1+\delta, 1+\epsilon)$ is approximately $1 + \delta\epsilon/2$. Note that if neither x or y is exactly 1, then $f(x, y)$ is not exactly 1, but when $f(x, y)$ is computed in double-precision arithmetic the result could be 1, and this is likely to happen when decoding a large polar code. This is a problem because to make bit decisions, we compare an LR to 1.

To avoid this problem, C. Leroux et al. [12] suggested computing the logarithms of the LRs (LLRs) instead of the LRs themselves. Then we make bit decisions by comparing LLRs to 0. This is better because a double-precision LLR can be extremely close to 0. Such LLRs correspond to LRs extremely close to 1, which cannot be distinguished from 1 in double precision.

A hardware implementation of a decoder would usually compute with fixed-point numbers instead of double precision. Fixed-point numbers cannot distinguish numbers near 0 as well as double precision numbers, but even so it is better to compute LLRs rather than LRs. The reason for this is that when decoding for a BMS channel, the LRs x and $1/x$ are equally likely to occur.

For example, if we are using LRs and are limited to 6 bits, we probably would choose to represent LRs from $1/8$ to 8 in steps of $1/8$ since $1/8$ and 8 are equally likely. This means we have 56 different LRs greater than 1 (probable zeroes), and only 7 less than 1 (probable ones). In contrast, we could represent

LLRs from -2 to $31/16$ in steps of $1/16$. These correspond to LRs from 0.135 to 6.94, roughly the same range, with a good balance between probable zeros and probable ones, and slightly better resolution near the decision point.

Reference [12] showed that in decoding a (1024, 512) polar code, using 6-bit LLRs resulted in performance very close to floating point. did not specify The range of LLRs these 6 bits represented is not specified in [12].

When we compute with LLRs, the formula for $f(x, y)$ becomes

$$\log\left(\frac{\exp(x)\exp(y)+1}{\exp(x)+\exp(y)}\right).$$

If we use this formula as written when either x or y is near 0, it causes the same loss of precision that results from computing with LRs near 1. Reference [10] gives the equivalent formula

$$f(x,y) = 2\tanh^{-1}\left(\tanh\left(\frac{x}{2}\right)\tanh\left(\frac{y}{2}\right)\right),$$

which is accurate near 0. The function $g(x, y, 0)$ becomes $x + y$, and $g(x, y, 1)$ becomes $y - x$. Reference [10] points out that these functions were already used for belief propagation decoding of LDPC codes. f is often called the box operator, and is often replaced by an approximate formula $\operatorname{sign}(x)\operatorname{sign}(y)\min(|x|, |y|)$. Reference [12] showed that this approximation caused negligible performance degradation for $N = 1024$, and about 0.1 decibels (dB) performance degradation for $N = 2^{14}$.

Version 3 of polarDecode.m provides options to approximate the box operator and to compute in fixed-point. These features were removed in version 4 for ease of use.

6.5 FEASIBLE BLOCK SIZES

Because [1] proved that better results can be achieved with large block sizes, it is of interest to determine what block sizes can be implemented in practice.

Many papers, including [12–17] have proposed implementing polar coding in field programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). Such designs can be much faster than software implementations, enabling large block sizes at high data rates.

Reference [14] describes an FPGA implementation that scales up to $N = 2^{21}$. The limiting factor is the amount of SRAM on the chip. Table VIII of [14] shows that block size 2^{21} requires 39,853,440 bits, which is available on an Altera Stratix® V 5SGXMB6R3F43I4 FPGA. The maximum data rate of this design is $\rho = 33\frac{K}{2^{21}}$ Mpbs. Therefore, it decodes a block in $T_d = \frac{2^{21}}{33 \cdot 10^6}\text{ s} = 64$ ms. With a code rate of $1/2$ and data rate of 10 Mbps, the total latency is about 270 ms.

Using the formula from [14], but different numbers of quantization bits ($Q_c = 6$, $Q = 4$), we found that block size 2^{22} requires 48,236,032 bits of SRAM, which is available on some Stratix® V models. Decoding a block requires 12.3 million clock cycles. We cannot predict T_d because the clock speed must be determined through FPGA synthesis.

The largest block size we have found in the polar codes literature was 2^{23}, in [18]. This code size was simulated in software. The throughput was not specified.

7. PERFORMANCE ANALYSIS

7.1 VIRTUAL CHANNELS

Figure 3 showed that an information-theoretic channel can encompass modulation and demodulation in addition to RF propagation. We can go further, and include the encoder and decoder in the channel, or parts of them, as shown in Figure 12. This figure shows a channel whose input is u_i for some i between 1 and N, and its output is the information that the decoder uses to determine $\hat{u}_i$, namely $\mathbf{y}$ and $\hat{u}_1$ through $\hat{u}_{i-1}$. (The set of possible output symbols is $\mathcal{Y}^N \times \{0,1\}^{i-1}$.) This channel represents how difficult it is to correctly decode u_i.

Reference [1] describes an *genie-aided decoder* that has access to the true values of u_1 through u_{i-1}, but not u_i, at the time it decides $\hat{u}_i$. The genie-aided decoder uses u_1 through u_{i-1} the same way the real decoder uses $\hat{u}_1$ through $\hat{u}_{i-1}$. The genie-aided decoder is easier to analyze than the real decoder, and it has a very useful property:

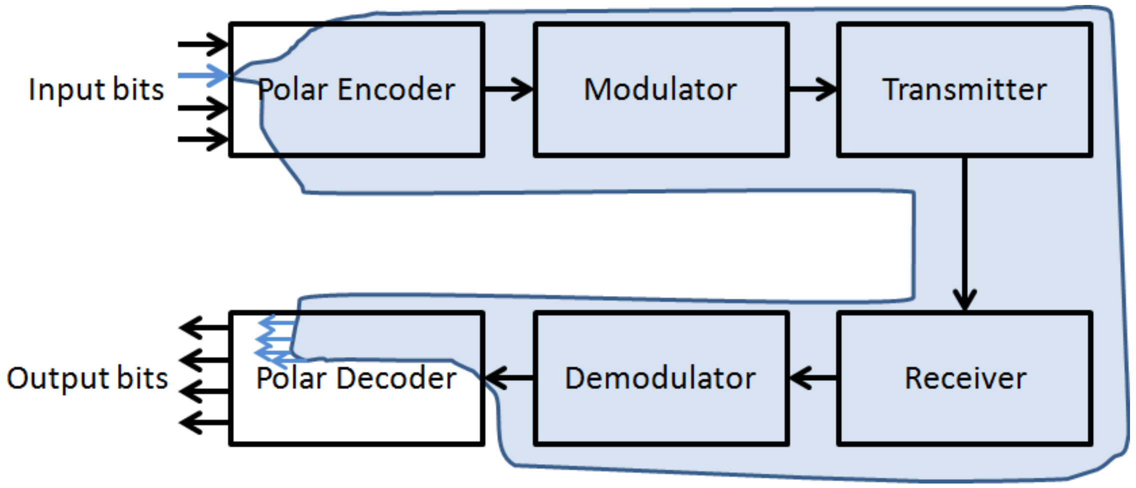

Figure 12. Virtual channel.

Proposition 2. *The genie-aided decoder will decode a block correctly if and only if the real decoder does.*

Proof. If either decoder makes a mistake, let u_i be the first bit that either decoder decodes incorrectly. Since the real decoder did not make a mistake before the ith bit, we have $\hat{u}_j = u_j$ for $1 \leq j < i$. So both decoders use the same input to decode u_i, so they produce the same answer. $\square$

Therefore, by analyzing the genie-aided decoder, we can prove results about the BLER of the real decoder.

For $1 \leq i \leq N$, the *virtual channel* $W_N^{(i)}$ is constructed from a channel W, a length N polar encoder, and a length N SC polar decoder. It has input u_i, and output $(\mathbf{y}, u_1, \ldots, u_{i-1})$. (Again, the set of possible output symbols is $\mathcal{Y}^N \times \{0,1\}^{i-1}$.) This channel represents how difficult it is for the genie-aided decoder to correctly decode u_i.

A virtual channel can be represented by an array with two rows, corresponding to the inputs 0 and 1, and one column for each output symbol. The entries in the array are the transition probabilities, and by definition each row must sum to 1. For example, if the outputs of W are $\mathcal{Y} = \{1,2,3,4,5,6,7,8\}$, then $W_4^{(3)}$ has $8^4 2^2 = 16384$ different output symbols ranging from $(1,1,1,1,0,0)$ to $(8,8,8,8,1,1)$.

However, recall from Section 6.2 that when decoding u_3, the decoder does not use six separate values y_1 through y_4, $\hat{u}_1$, and $\hat{u}_2$. Instead, it uses the LR of u_3, which is computed recursively from those six values. This LR is the ratio of the two numbers in one column of the array. So we may think of these 16384 output symbols as merely 16384 different columns in an array; we do not need to know if a particular column represents $(2, 1, 1, 7, 0, 0)$ or $(5, 4, 3, 5, 1, 0)$, for example. Also, if $\begin{bmatrix} a \\ b \end{bmatrix}$ and $\begin{bmatrix} c \\ d \end{bmatrix}$ are two columns such that $a/b = c/d$, i.e., the two columns have the same LR, they are effectively identical, and may be replaced by a single column $\begin{bmatrix} a+c \\ b+d \end{bmatrix}$. [5]

The channel probabilities of the virtual channels give the probabilities of the various LRs that can arise in the decoder. For example, suppose one of the columns in $W_4^{(3)}$ is $\begin{bmatrix} 0.2 \\ 0.1 \end{bmatrix}$. This column means that in 10% of all uses of the length 4 genie-aided decoder, $u_3 = 0$ and $L(u_3) = 0.2/0.1 = 2$. Likewise, in 5% of all uses of the length 4 genie-aided decoder, $u_3 = 1$ and $L(u_3) = 2$. (We multiply the array numbers by 0.5 because u_3 has a 50% chance of being 0 and a 50% chance of being 1.) In both cases, the genie-aided decoder decides $\hat{u}_3 = 0$, so this column contributes 0.05 to the BER of u_3. We compute the total BER of u_3 by adding the smaller entry from each column, and dividing the sum by 2.

7.2 CHANNEL POLARIZATION

For any $n \geq 0$, $N = 2^n$, and $1 \leq i \leq N$, Equation (22) of [1] shows how to compute the transition probabilities of $W_{2N}^{(2i-1)}$ from the transition probabilities of $W_N^{(i)}$. Similarly, Equation (23) shows how to compute the transition probabilities of $W_{2N}^{(2i)}$ from the transition probabilities of $W_N^{(i)}$.

Let $I(Q)$ denote the capacity of a channel Q,[6] and let $Q \mapsto Q'$ and $Q \mapsto Q''$ denote the transformations described in Equations (22) and (23), respectively. Proposition 4 of [1] is that $I(Q') + I(Q'') = 2I(Q)$, and $I(Q') \leq I(Q) \leq I(Q'')$, with equality if and only if $I(Q)$ equals 0 or 1.

Starting with W, we can compute $W_N^{(i)}$ for any i between 1 and N by using Equations (22) and (23) n times. There are N different ways to choose a sequence of n transformations, and these N sequences yield N different virtual channels $W_N^{(1)}$ through $W_N^{(N)}$, as illustrated in Figure 6 of [1], "The tree process for the recursive channel construction."

Arikan proved (Theorem 1 of [1]) that for large enough n, almost all of the virtual channels have capacity near 0 or near 1. This effect is called *channel polarization*. If a BMS channel has capacity near 1, its BER is near 0, and if its capacity is near 0, its BER is near 0.5. Furthermore, the proportion of virtual channels with capacity near 1 is close to $I(W)$. Therefore, we can construct a code with rate near $I(W)$ and a low BER by freezing all inputs i such that $I(W_N^{(i)})$ is not near 1.

[5]Proposition 13 of [1] shows that if W is symmetric, then so is $W_N^{(i)}$. If we combine two columns, then we should also combine their complements to preserve the symmetry.

[6]Throughout this document, we use W for a channel that gets its input from an encoder and sends its output to a decoder. $W_N^{(i)}$ always represents a virtual channel constructed from W. We use Q for an arbitrary channel that could be either of these types or something else.

8. CODE CONSTRUCTION

We construct an (N, K) polar code by choosing K elements of $\{1, \ldots, N\}$ to be the non-frozen indices. To get a good code for a particular BMS channel W, we must choose indices i such that $I(W_N^{(i)})$ is near 1. However, computing $I(W_N^{(i)})$ exactly is not feasible for even moderate code sizes, because if W has a outputs, then $W_N^{(i)}$ has $a^N 2^{i-1}$ outputs. Various approaches have been proposed for estimating the capacity or BER of $W_N^{(i)}$.

8.1 TAL/VARDY METHOD

We consider [19] by I. Tal and A. Vardy as the state of the art in code construction. Their method uses a parameter μ. A larger μ means a better approximation but also longer running time. To construct a code of length N, the algorithm uses $O(N\mu^2 \log \mu)$ instructions. Reference [19] gives results for values of μ ranging from 8 to 512. The method works by starting with the transition probabilities of W, and then constructing virtual channels following the tree process shown in Figure 6 of [1]. Reference [19] uses $Q \mapsto Q \boxast Q$ and $Q \mapsto Q \circledast Q$ for the channel transformations described in Equations (22) and (23) of [1].

Whenever a channel is constructed with $> \mu$ outputs, it is replaced by a similar channel with μ outputs. At each leaf of the tree, we produce a channel Q_i that is similar to $W_N^{(i)}$, and we compute the BER of Q_i to estimate the BER of $W_N^{(i)}$.

One might ask how a channel with a huge number of outputs could be similar to a channel with a small number of outputs. The answer is that the decoder only cares about the LR of each output. If we have many symbols with approximately equal LRs, we can combine them to one symbol whose LR is within the range of those LRs, and this change has very little effect on the channel's BER.

Reference [19] presents two methods to reduce the number of channel outputs to μ. The first method is called "degrading merge". It is used in "Algorithm A", which guarantees that $\mathrm{BER}(Q_i) \geq \mathrm{BER}(W_N^{(i)})$. The second method is called "upgrading merge". It is used in "Algorithm B", which guarantees that $\mathrm{BER}(Q_i) \leq \mathrm{BER}(W_N^{(i)})$. Thus $\mathrm{BER}(W_N^{(i)})$ is bounded between two values.[7] If these values are far apart, it may be worthwhile to rerun the algorithms with a larger μ to get a better estimate. Reference [19] gives examples showing that the bounds can be close together for large enough μ. It also presents "Algorithm D", which is the same as Algorithm A except that it computes an additional number Z_i that is guaranteed to satisfy $\mathrm{BER}(Q_i) \geq Z_i \geq \mathrm{BER}(W_N^{(i)})$, giving an even better upper bound.

After running Algorithm A (resp. D), we construct an (N, K) polar code by letting $\mathcal{A}$ be the indices of the K smallest values of $\mathrm{BER}(Q_i)$ (resp. Z_i). The sum of these K values is an upper bound on the BLER of this code when used for channel W. K could be chosen in advance, or it could be chosen to achieve a particular BLER bound.

8.1.1 Implementation

The file algorithmA.m is a MATLAB® implementation of Algorithm A, and AlgorithmD.java is a Java™ implementation of Algorithm D. We did not implement Algorithm B. Reference [19] also shows how to modify the degrading merge algorithm to convert a continuous channel to a similar discrete channel. This algorithm assumes that we are able to compute integrals of the transition probabilities. We im-

[7]These guarantees depend on exactly implementing the operations described in the paper. If these operations are implemented with double precision arithmetic, it is possible that accumulated rounding error could cause violation of the bounds.

plemented the continuous degrading merge for the specific case of an AWGN channel. The MATLAB® version is degradeAwgnToDiscrete.m, and the Java™ version is a constructor in Channel.java.

The function `algorithmA` returns the capacity, BER, and Bhattacharyya parameter (see [1] for definition) of each Q_i. The user can then use other MATLAB® functions to sort the BERs and choose $\mathcal{A}$.

In the Java™ implementation, the `AlgorithmD` class has a `compute` method that estimates $\mathrm{BER}(W_{2^j}^{(i)})$ for $1 \le j \le m$ and $1 \le i \le 2^j$. This class also has a `computeAndSaveFixedRateCodes` method. For each j this method sorts the 2^j BER estimates, chooses $\mathcal{A}$ to construct a length 2^j code of a specified rate, and computes the bound for this code's BLER. Finally, this method creates a MATLAB® script file. When this script is run in MATLAB®, it creates one MAT-file for each code. The individual BER estimates are not available to MATLAB®, but they are saved by serialization of the `AlgorithmD` object.

The degrading merge algorithm using an array with L columns to represent a channel with $2L$ outputs. Specifically, if $Q(y|0) = a$ and $Q(y|1) = b$, then $Q(\bar{y}|0) = b$ and $Q(\bar{y}|1) = a$. If $a \ge b$, then the column $\begin{bmatrix} a \\ b \end{bmatrix}$ explicitly represents y and implicitly represents $\bar{y}$.

If $\mathcal{Y}$ is the set of outputs of Q, then the outputs of $Q \boxast Q$ are $\mathcal{Y} \times \mathcal{Y}$, and the outputs of $Q \boxast Q$ are $\mathcal{Y} \times \mathcal{Y} \times \{0, 1\}$.

Here are some things we learned in the process of coding the algorithms:

1. Reference [19] gives no details of how to implement $Q \boxast Q$ and $Q \circledast Q$. We found that both of these operations could be implemented more easily using an array that has one column for each *pair* of complementary channel outputs. For $\boxast$, if the input has L columns representing $2L$ channel outputs, then the output has $2L^2$ columns representing $4L^2$ channel outputs. We had to ensure that these $2L^2$ columns included one of each pair of complementary channel outputs. We found that in $Q \boxast Q$, the complement of channel output (y_1, y_2) is $(y_1, \bar{y}_2)$. So the `boxStar` function computes $(Q \boxast Q)(y_1, y_2|u)$ for all $2L$ values of y_1, but only L values of y_2, namely those explicitly represented in the input.

2. For $\circledast$, if the input has L columns representing $2L$ channel outputs, then the output has $4L^2$ columns representing $8L^2$ channel outputs. However, we found that for all $y_1, y_2 \in \mathcal{Y}$ and $u_2 \in \{0, 1\}$, $(Q \circledast Q)(y_1, y_2, 1|u_2) = (Q \circledast Q)(\bar{y}_1, y_2, 0|u_2)$. This means that channel outputs $(y_1, y_2, 1)$ and $(\bar{y}_1, y_2, 0)$ are equivalent, and may be combined. This reduces $Q \circledast Q$ to $2L^2$ columns representing $4L^2$ channel outputs. The `circleStar` function computes $(Q \circledast Q)(y_1, y_2, 0|u_2)$ only, and doubles the results to include the symbols combined with them.

3. We had to ensure that these $2L^2$ columns included one of each pair of complementary channel outputs. We found that in $Q \circledast Q$, the complement of channel output $(y_1, y_2, 0)$ is $(\bar{y}_1, \bar{y}_2, 0)$. So the `circleStar` function computes $(Q \circledast Q)(y_1, y_2, 0|u)$ for all $2L$ values of y_1, but only L values of y_2, namely those explicitly represented in the input.

4. Recall that if channel output y has transition probabilities $\begin{bmatrix} a \\ b \end{bmatrix}$, then $\bar{y}$ has transition probabilities $\begin{bmatrix} b \\ a \end{bmatrix}$. The `boxStar` or `circleStar` function will work correctly if its input includes one of each pair y, $\bar{y}$; it does not matter which one is included. However, the degrading merge requires its input to include the one with the larger probability on top. The outputs of `boxStar` and `circleStar` generally do not satisfy this property, even if the input does. So whenever `boxStar` or `circleStar`

outputs a column $\begin{bmatrix} a \\ b \end{bmatrix}$ with $a < b$, we replace this column with $\begin{bmatrix} b \\ a \end{bmatrix}$, which represents the same pair of complementary channel outputs.

5. The degrading merge works by combining channel outputs. Two columns $\begin{bmatrix} a \\ b \end{bmatrix}$ and $\begin{bmatrix} c \\ d \end{bmatrix}$ with similar LRs are replaced by a single column $\begin{bmatrix} a+c \\ b+d \end{bmatrix}$. The degrading merge algorithm chooses which columns to combine with the goal of minimizing the decrease in channel capacity.

   One might ask, since we are ultimately interested in the BER of each virtual channel, why do we minimize the decrease in channel capacity, instead of minimizing the increase in BER? We discovered that combining columns of Q has *no* effect on the BER of Q, but it can increase the BER of $Q \circledast Q$ and other channels that occur later in the tree. Reference [19] does not explain how to compute this effect, so it does not prove that this method of combining columns minimizes the increase in BER.

6. The degrading merge algorithm uses a heap data structure to determine which columns to combine. Reference [19] defines a heap as a data structure that associates keys to data, and supports four operations:

   - Insert: add a new key-value pair to the heap
   - GetMin: find the key-value pair with the minimum key in the heap
   - RemoveMin: remove the key-value pair with the minimum key in the heap
   - ValueUpdated: change the key of a key-value pair.

   It states that getMin has running time $O(1)$, and the other three operations have running time $O(\log L)$ where L is the size of the heap.

   Such a structure is often called a min-heap, to distinguish it from a max-heap that supports getMax and removeMax instead of getMin and removeMin.

   Reference [19] refers to [20] for the heap implementation. However, the heap described in [20] does not support the valueUpdated operation. We searched the Internet for other heap implementations, and found that max-heaps typically support increaseKey and min-heaps typically support decreaseKey, but we could not find an implementation that supports both. Fortunately, we found a way to implement decreaseKey on a max-heap in $O(\log L)$ time.

8.1.2 Results

Table I of [19] shows some results of computing a $(2^{20}, 445340)$ polar code for the channel BSC($p = 0.11$). It shows that using Algorithm A with $\mu = 8$, the resulting BLER upper bound is 5.096030e-03. Using our implementation, we obtained 5.083668e-03, which is about 0.2% smaller.

Figure 2(b) of [19] shows some results of computing polar codes for the channel AWGN($\sigma^2 = 0.1581$). The code lengths are 2^{10} and 2^{20}, and for each length the code rates range from 0.9 to 1. Reference [19] says "the value of μ did not exceed 512." This figure is a graph with code rate on the horizontal axis, and the base-10 logarithm of the BLER bound on the vertical axis.

We used AlgorithmD.java to repeat these calculations for the $N = 2^{10}$ case using $\mu = 512$. Figure 13 shows a graph of our results. The results appear to be very similar to Figure 2(b) of [19].

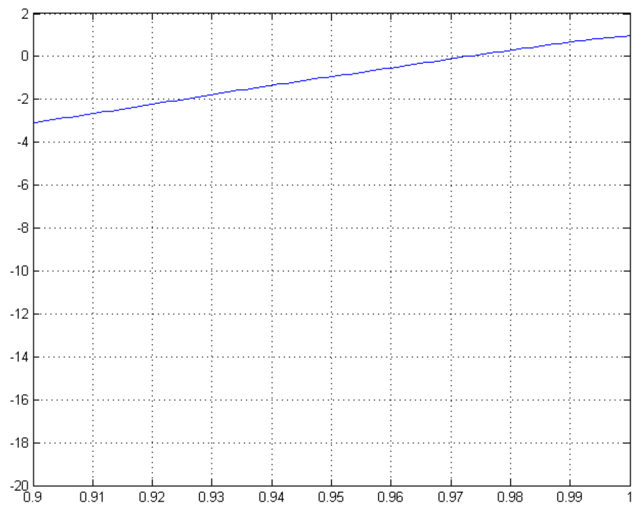

Figure 13. Reproducing a result from Tal and Vardy.

8.2 GAUSSIAN APPROXIMATION METHOD FOR AWGN CHANNELS

Recall from Section 7.1 that the transition probabilities of the virtual channels indicate the probabilities of the various LRs that can arise in the decoder. A column $\begin{bmatrix} a \\ b \end{bmatrix}$ represents a probability $(a + b)/2$ that the LR equals a/b. The Tal/Vardy code construction method works by approximating the virtual channel, using a channel with at most μ outputs. So the approximate LR distribution is described by a set of 2μ numbers, half of which may be omitted due to symmetry.

Several papers, including [6] and [21], have suggested that for an AWGN channel we can use a more efficient method. If we describe the distribution of LLRs instead of LRs, we can use a much simpler approximation: a Gaussian distribution. However, neither of these papers clearly explains why this method works. We found the explanation in [22] by D. Wu, Y. Li, and Y. Sun. However, their explanation requires a correction.

Previously, we have used L to represent an LR, but in this section it will be an LLR. Let $L_N^{(i)}$ denote the LLR of $W_N^{(i)}$, i.e., the LLR of the u_i computed by a genie-aided SC decoder. $L_N^{(i)}$ is a random variable that depends on $\mathbf{y}$ and u_1 through u_{i-1}. $L_N^{(2i-1)}$ and $L_N^{(2i)}$ are both computed recursively from two other random variables that each have the distribution of $L_{N/2}^{(i)}$, using the functions f and g described in Section 6.4.

It is well known that when using optimal decoding of a linear block code with a BMS channel, the probability of block error is the same regardless of which codeword was transmitted.[8] The SC polar decoder is not optimal, but Arikan proved something similar for SC decoding:

Theorem 3. *(Corollary 1 of [1]) For $1 \leq i \leq N$, the genie-aided decoder's probability of incorrectly decoding u_i is the same regardless of which codeword is transmitted.*

Therefore, we can assume for convenience that $\mathbf{u}$ and $\mathbf{x}$ are all zeroes. To see why this is helpful, refer to the decoding procedure in Section 6.3. In the genie-aided decoder, it is the u_i's, not the $\hat{u}_i$'s, that propa-

[8]This assumes that ties are broken randomly. In an AWGN channel (or any continuous channel), the probability of an exact tie is 0, so we will ignore the possibility of ties in this section.

gate from left to right through the decoder. So when we compute $L(w_{2i}) = g(L(x_{2i-1}), L(x_{2i}), \hat{v}_i)$ in step 5, $\hat{v}_i$ is always 0, so g is always an addition, not a subtraction.

Now let W be an AWGN channel. Recall that its transition probabilities are $W(y|0) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(y-1)^2}{2\sigma^2})$ and $W(y|1) = \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(y+1)^2}{2\sigma^2})$. For any output y, the corresponding LLR is $\log\left(\frac{W(y|0)}{W(y|1)}\right)$, and since we assume only 0's are transmitted, the probability of this LLR occurring is $W(y|0)$. We compute

$$L(y) = \log\left(\frac{W(y|0)}{W(y|1)}\right) = \log\left(\frac{\frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(y-1)^2}{2\sigma^2})}{\frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{(y+1)^2}{2\sigma^2})}\right)$$

$$= \log\left(\exp\left(-\frac{(y-1)^2}{2\sigma^2} + \frac{(y+1)^2}{2\sigma^2}\right)\right) = \frac{-(y-1)^2 + (y+1)^2}{2\sigma^2} = \frac{2y}{\sigma^2}.$$

Since y is a normally distributed variable with mean 1 and variance σ^2, then $L(y) = \frac{2y}{\sigma^2}$ is a normally distributed variable with mean $2/\sigma^2$ and variance $\sigma^2\left(\frac{2}{\sigma^2}\right)^2 = 4/\sigma^2$. The variance is twice the mean. We define a *consistent normal density* variable (CNDV) to be a normally distributed variable whose variance is twice its mean. Then $L(y)$ is a CNDV.

Next, $L_2^{(1)} = f(L(y_1), L(y_2))$, where $L(y_1)$ and $L(y_2)$ are both variables with the distribution of $L(y)$. According to the "GA principle," when f is applied to two CNDVs, the result can be approximated by a CNDV. Therefore, $L_2^{(1)}$ is an approximate CNDV. In addition, $L_2^{(2)} = g(L(y_1), L(y_2), 0) = L(y_1) + L(y_2)$. It is well known that the sum of normal variables is a normal variable. The mean of the sum is the sum of the means, and the variance of the sum is the sum of the variances. So $L_2^{(2)}$ has mean $4/\sigma^2$ and variance $8/\sigma^2$, and is a CNDV. Applying these arguments repeatedly, we find that all of the virtual channels are approximate CNDVs. However, it is not clear how much the approximation degrades when f is applied repeatedly.

Since each $L_N^{(i)}$ is normally distributed with the variance equal to twice the mean, we can characterize its distribution using the mean only, compared to μ parameters used in the Tal/Vardy method. We write this mean as $E(L_N^{(i)})$. We compute it recursively: $E(L_N^{(2i)}) = 2E(L_{N/2}^{(i)})$, and $E(L_N^{(2i-1)}) = \omega\left(E(L_{N/2}^{(i)})\right)$, where ω is defined by

$$\omega(x) = \phi^{-1}\left(1 - (1 - \phi(x))^2\right) \tag{2}$$

and ϕ is defined by

$$\phi(x) = 1 - \frac{1}{\sqrt{4\pi x}} \int_{-\infty}^{\infty} \tanh\left(\frac{\tau}{2}\right) \exp\left[-(\tau - x)^2/(4x)\right] \, d\tau. \tag{3}$$

Finally, $\mathrm{BER}(W_N^{(i)}) = \frac{1}{2}\,\mathrm{erfc}\left(0.5\sqrt{E(L_N^{(i)})}\right)$.

8.2.1 Error in Wu, Li, and Sun

The procedure above computes an estimate of $\mathrm{BER}(W_N^{(i)})$, the probability of error of a genie-aided decoder. However, [22] claims that this procedure computes $P(\mathcal{C}_i)$, defined as the conditional probability of a real (i.e., not genie-aided) SC decoder incorrectly decoding the ith bit, subject to the condition that all previous bits are decoded correctly. The BLER is then computed as $1 - \prod_{i\in\mathcal{A}}(1 - P(\mathcal{C}_i))$.

At first glance, one might think that $P(\mathcal{C}_i) = \mathrm{BER}(W_N^{(i)})$ because both are the probability of error in a situation where the decoder has the correct value of all previous bits. However, $\mathrm{BER}(W_N^{(i)})$ considers all possible values of $\mathbf{y}$, but $P(\mathcal{C}_i)$ considers only values of $\mathbf{y}$ that would have caused all previous bits to be decoded correctly. This imposes a selection bias on the channel noise. If several previous bits were all decoded correctly, this probably means that not many bits were flipped in the channel, which means that u_i is also more likely to be decoded correctly. Therefore, we expect $P(\mathcal{C}_i) < \mathrm{BER}(W_N^{(i)})$, although this is not a proof.

$P(\mathcal{C}_i)$ is affected by frozen bits. If the ith bit is frozen then $P(\mathcal{C}_i) = 0$, so assume the ith bit is not frozen. If some of the first $i-1$ bits are frozen, then this enlarges the set of vectors y_1^N that will cause the first $i-1$ bits to all be decoded correctly, so the bias is reduced. If the first $i-1$ bits are all frozen, then $P(\mathcal{C}_i) = \mathrm{BER}(W_N^{(i)})$.

For example, let $N = 2$, and let $\sigma^2 = 0.25$. Using the algorithm of [22], we computed the estimates $P(\mathcal{C}_1) = 0.04473$ and $P(\mathcal{C}_2) = 0.002363$. We then ran 10^6 trials of encoding, transmission, and genie-aided decoding with no frozen bits, producing the estimates $P(\mathcal{C}_1) = \mathrm{BER}(W_2^{(1)}) = 0.044794$, $\mathrm{BER}(W_2^{(2)}) = 0.002304$, and $P(\mathcal{C}_2) = 5.4217 \cdot 10^{-4}$. The corresponding 3-sigma confidence intervals are $(0.044173, 0.045415)$, $(0.002160, 0.002448)$, and $(4.5715 \cdot 10^{-4}, 5.9812 \cdot 10^{-4})$. So the $P(\mathcal{C}_i)$'s computed with the algorithm agree with the simulation's $\mathrm{BER}(W_2^{(i)})$'s.

If we had an efficient way to compute $P(\mathcal{C}_i)$, it would still be difficult to use this for code construction because of the dependence on previous frozen bits.

8.2.2 Implementation

The file wuLiSun.m is our MATLAB® implementation of the Gaussian approximation method.

The function ϕ is hard to compute because it requires integrating from $-\infty$ to ∞. Reference [22] uses an approximation:

$$\phi(x) = \begin{cases} \exp(-0.4527x^{0.86} + 0.0218), & 0 < x < 10, \\ \sqrt{\frac{\pi}{x}} \exp\left(-\frac{x}{4}\right)\left(1 - \frac{10}{7x}\right) & x \geq 10. \end{cases} \tag{4}$$

Reference [22] then says "In practice, we can store samples of (4), and look up the result via a bi-search method." We took this to mean that the look-up table is used to compute ϕ^{-1}. The look-up table contains $\phi(x)$ for many values of x. To compute $\phi^{-1}(y)$, we find the value nearest y in the look-up table, and return the corresponding x. Reference [22] says nothing about how many values to include in the table or how to choose them. We used 10^8 values of x, logarithmically spaced from 2^{-100} to 2^{100}.

The file gaussianLlrXor.m is our MATLAB® implementation of the function ω. We found that for very small values of x, $\omega(x) > x$. This cannot be correct because it would imply that $Q \boxast Q$ has a higher capacity than Q. Apparently, it indicates that the approximation of ϕ is not good for very small x. So we modified gaussianLlrXor.m to return $\min(x, \omega(x))$. This is an improvement, but the results are still incorrect for very small x.

8.2.3 Results

To compare the two code construction methods, we let W be AWGN($\sigma^2 = 0.1581$), $N = 2^{10}$, and used both methods to compute $\mathrm{BER}(W_N^i)$ for $1 \leq i \leq N$. The function `wuLiSun` ran in 8 seconds, compared to 225 seconds for AlgorithmD.java. We also ran simulations of a genie-aided decoder in this channel for about eight hours, a total of 479453 trials, to produce empirical estimates of $\mathrm{BER}(W_N^i)$. We then tested if these were consistent with the estimates computed by the code construction methods. For 894 of the 1024 indices, the number of observed errors was < 3, and for all 894 of these, the computed BERs were reasonable. For the remaining 130 indices, we tested the hypothesis that the observed number of errors is a sample from the binomial distribution with $n = 479453$ and p equal to the computed BER.

We approximated the binomial distribution with a normal distribution with mean np and variance $np(1 - p)$. The resulting Z-scores are in Figure 14.

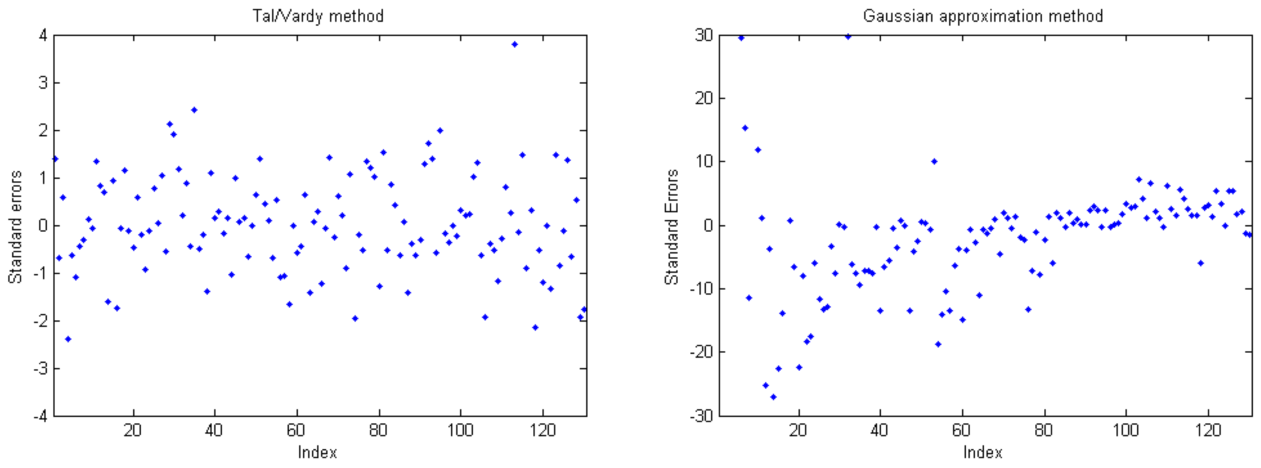

Figure 14. Z scores.

The Tal/Vardy BER estimates appear very reasonable, as 64% are within one standard error of the empirical estimates, and 96% are within two standard errors, which is roughly what we expect. Only the one , almost four standard errors above the mean, indicates a small inaccuracy in the method. In contrast, only 47% of the Gaussian approximation estimates are within three standard errors, and 75% are within ten standard errors. For seven of the indices (not shown), the empirical BER was more than 30 standard errors above the estimate.

On the other hand, if we are constructing a code with a predetermined rate, then this inaccuracy does not have a large effect. In Figure 15, the green line is the BLER bound computed with the Tal/Vardy method. The blue line is the BLER bound obtained by choosing $\mathcal{A}$ by the Gaussian approximation method, and then using the BER estimates from the Tal/Vardy method.

The difference is never more than 10% except for lower values of K where the BLER bound is less than 10^{-6}. The Gaussian approximation method may be useful if a code must be constructed quickly. However, for larger N or larger σ^2, very small LLRs will arise more often, so the inaccuracy of this method could have a larger effect. It may be necessary to compute $\omega(x)$ more accurately for very small values of x.

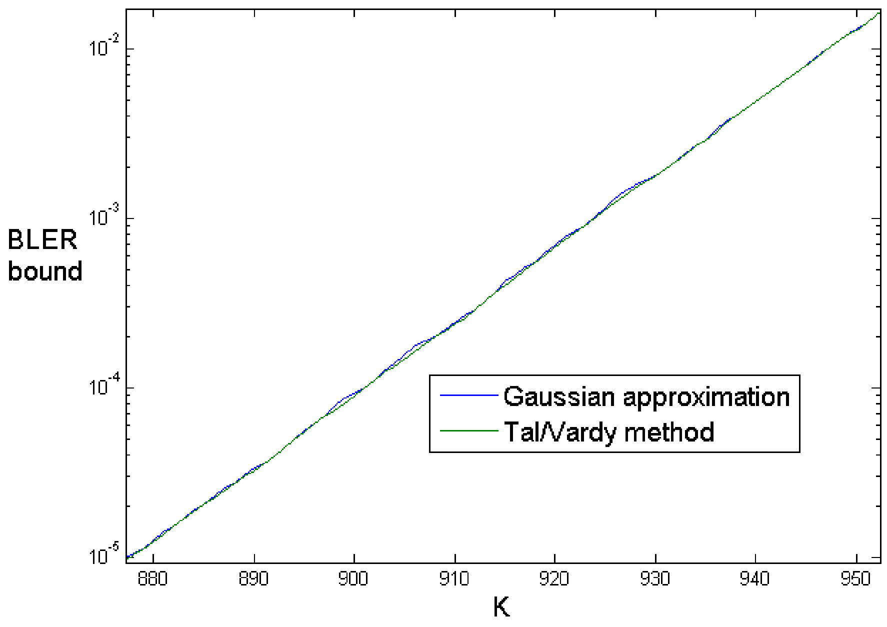

Figure 15. Comparison of code construction methods.

9. OTHER VERSIONS OF POLAR CODING

9.1 ENCODING WITH $F^{\otimes N}$

Recall that we defined the function G_N as $B_N F^{\otimes n}$. Section VII.C of [1] says "In an actual implementation of polar codes, it may be preferable to use $F^{\otimes n}$ in place of $B_N F^{\otimes n}$ as the encoder mapping in order to simplify the implementation."

Figure 16 shows two different ways to recursively compute $F^{\otimes n}$. The base case is $F^{\otimes 1} = F = G_2$. Figure 16(a) is a generalization of Figure 9 in [1].

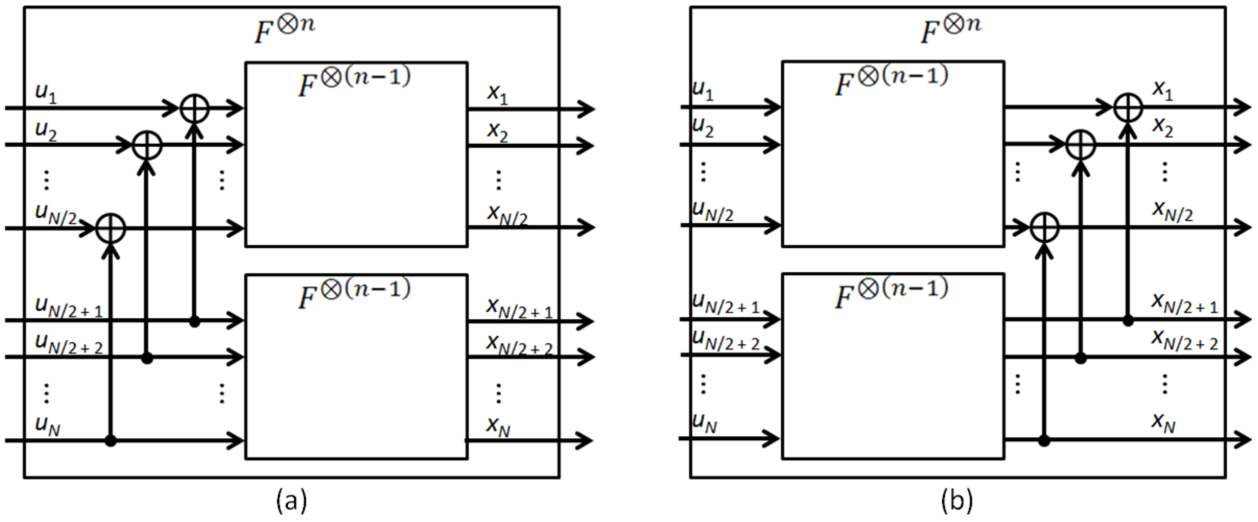

Figure 16. Two recursive constructions of $F^{\otimes n}$.

These encoders are also built from interconnected copies of G_2. For example, in Figure 16(a), u_1 and $u_{N/2+1}$ are the inputs to a G_2, u_2 and $u_{N/2+2}$ are the inputs to another G_2, etc. The total number of G_2's in either encoder is $nN/2$.

F_N.m is a MATLAB® implementation of $F^{\otimes n}$. It uses the structure in Figure 16(a).

The corresponding decoder is built using the same strategy described in Section 6.2: it has $nN/2$ decoding elements that each decode one copy of G_2, and the connections between these decoding elements parallel the connections between the G_2's in the encoder. We can use a decoder based on Figure 16(a) or (b) regardless of which structure the encoder follows, because both encoder structures produce the same output. For the same reasons given in Section 6.3, the structure in Figure 16(b) is better for implementing the decoder with a recursive function.

9.2 SYSTEMATIC POLAR CODING

E. Arikan introduced systematic polar coding in [23]. "Systematic" means that the output is embedded in the input. As before, the encoder uses the function G_N or $F^{\otimes n}$, and we divide the set $\{1, \ldots, N\}$ into two sets $\mathcal{A}$ of size K and $\mathcal{A}^C$ of size N_K. As before, the inputs of G_N (or $F^{\otimes n}$) are called u_1 through u_N, the outputs are called x_1 through x_N, and the $u(\mathcal{A}^C)$ are frozen. The difference is that the encoder's input goes into a preprocessor, which solves for the unique $u(\mathcal{A})$ that causes $x(\mathcal{A})$ to equal the encoder input. Figure 17 shows an $(8, 4)$ systematic encoder with $\mathcal{A} = \{4, 6, 7, 8\}$.

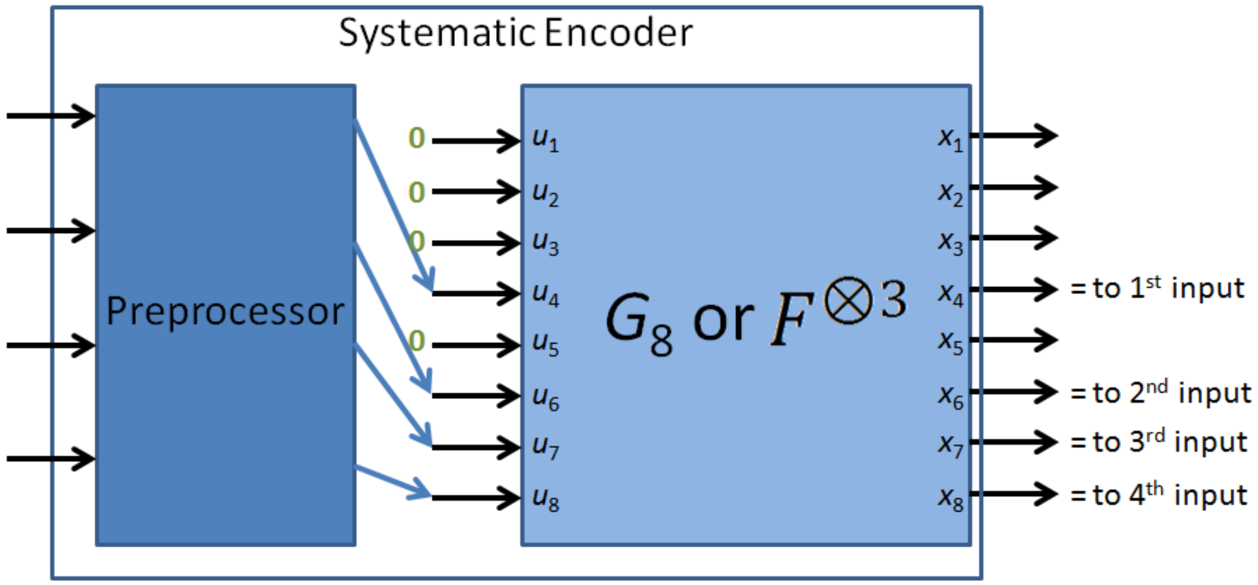

Figure 17. Systematic polar encoder.

The systematic polar decoder is exactly the same as the non-systematic decoder, except that at the end it returns $\hat{x}(\mathcal{A})$ instead of $\hat{u}(\mathcal{A})$.

Reference [23] presents simulation results showing that the systematic decoder has the same BLER and a lower BER than the non-systematic decoder. The lower BER is surprising, because both decoders decide $\hat{u}$ the same way. If errors are made in deciding $\hat{u}$, these errors propagate from left to right through the decoder, so one might expect to get even more errors at the right where $\hat{x}$ is decided. Instead, we find that errors cancel out, so $\hat{x}$ has fewer errors. This is an empirical result; as far as we know, it has never been proved or even explained.

The file systematicPolarEncode.m is a MATLAB® implementation of a systematic polar encoder using $F^{\otimes n}$ as shown in Figure 16(a). The file polarDecode.m is a MATLAB® implementation of a systematic polar decoder using $F^{\otimes n}$, based on an algorithm described in reference [24]. We created Figure 16(b) to match the structure of this algorithm.

10. SIMULATIONS

10.1 MEASURES OF PERFORMANCE

Recall that in the Gaussian channel described in Section 3, the BPSK modulator outputs ± 1, and the channel output is $\pm 1 + n$ where n is normally distributed with mean 0 and variance σ^2. The quantity $\frac{1}{2\sigma^2}$ is often called the *bit energy to noise power spectral density ratio* (E_b/N_0), and is usually expressed in decibels.

However, when FEC coding is used, E_b is redefined as the energy received per *information* bit, as opposed to the *symbol energy* (E_s), which is the amount of energy received in one use of the channel. Using an (N, K) code, K bits of information are sent in N uses of the channel, so $KE_b = NE_s$. We have $E_s/N_0 = \frac{1}{2\sigma^2}$, and $E_b/N_0 = \frac{N}{2\sigma^2 K}$. E_s/N_0 is a property of the channel, but E_b/N_0 depends on the code used with that channel.

In the polar codes literature, most reported simulation results are for Gaussian channels, and these results are usually displayed with a graph called a *waterfall plot*. In these graphs, the vertical axis usually shows BER or BLER on a logarithmic scale, and the horizontal axis usually shows E_b/N_0 in decibels. Figure 1 of [23] is a good example. This graph compares the performance of a systematic polar code with the corresponding non-systematic code, so the graph has one line for each code. Each of these codes was tested in 13 different Gaussian channels, resulting in 13 points on each line.

10.2 ESTIMATING BER

We wrote the MATLAB® function `codeTest2` for simulating polar codes in Gaussian channels and estimating their BERs. It repeatedly performs the following steps:

1. Generate K random bits.
2. Input those bits to the encoder, which outputs N bits.
3. Map these N bits to ± 1 and add normally distributed noise.
4. Compute the LLRs of the results.
5. Input these N LLRs to the decoder, which outputs K bits.
6. Compare the output bits to the original random bits, and count errors.

One difficulty is deciding when to stop. A common rule of thumb in digital communications is to stop a test when 100 bit errors have occurred ([25]). However, this rule is based on the assumption that each bit is incorrect with probability p, independent of all other bits. This assumption is incorrect when FEC coding is used. We might decode several blocks with no errors, and then get more than 100 bit errors in a single block. This tells us little about how frequently block errors will occur, and how many bit errors will be included in each block error. Figure 18 is a histogram showing various numbers of bit errors found in one block error.

Suppose we decode n blocks and get m block errors. We keep track of the number of bit errors in each of these m blocks: $x_1, x_2, \ldots, x_m$. Let $\bar{x}$ be the mean of $\{x_1, x_2, \ldots, x_m\}$ and let s be the standard deviation. Then the observed BER is $\frac{m}{n}\frac{\bar{x}}{K}$. $\frac{m}{n}$ is the observed BLER and we will call $\frac{\bar{x}}{K}$ the observed *bit error*

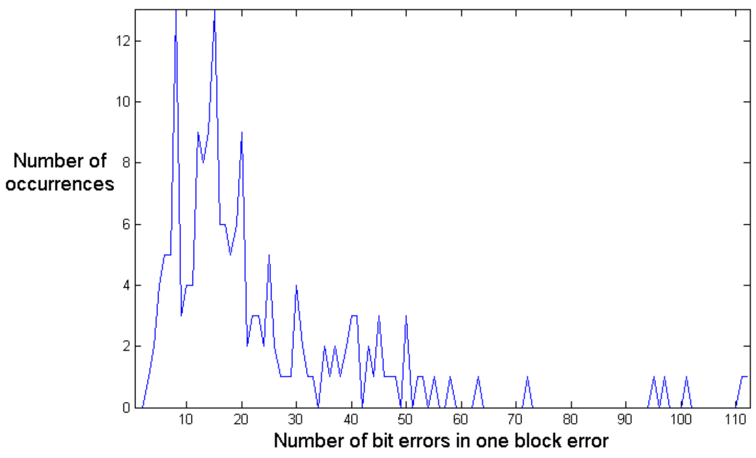

Figure 18. 177 block errors in a (1024, 512) polar code with E_b/N_0 = 2.5 decibels.

rate of erroneous blocks (BEREB). We can estimate BLER and BEREB separately. Since we plot BER on a logarithmic scale, our metric is the relative error

$$\frac{(\text{true BER}) - (\text{observed BER})}{(\text{observed BER})}.$$

Our goal is that the standard error of our estimates should be about 10%. We can make an exception for very low BERs. A commonly used target BER is 10^{-5} (e.g., in Table XVI of [26]), so if we are confident that a BER is below 10^{-5} we are not concerned with the relative error.

The relative error of BER is the product of the relative errors of BLER and BEREB. For small errors these are roughly additive, e.g., if we overestimate BLER by 3% and BEREB by 5%, then we overestimate BER by about 8%. So if the standard errors of our estimates of BLER and BEREB are $a\%$ and $b\%$, respectively, then the standard error of our BER estimate is about $\sqrt{a^2+b^2}\%$.

If the true BLER is p, then m will have a binomial distribution with mean np and standard deviation $\sqrt{np(1-p)}$. For small p, this is roughly $\sqrt{np}$. Therefore, if we have observed m block errors, the standard error of BLER is about $\sqrt{m}$, and relative standard error is about $1/\sqrt{m}$.

If $m \geq 30$, then $\bar{x}/K$ will have an approximately normal distribution with standard deviation $s/\sqrt{m}$, so the standard error of our BEREB estimate is

$$\frac{sK}{\bar{x}\sqrt{m}}.$$

For $1 \leq m < 30$, this formula is less accurate and tends to underestimate the standard error. The function `codeTest2` roughly compensates by using $\sqrt{m-1}$ in the denominator in place of $\sqrt{m}$:

$$\frac{sK}{\bar{x}\sqrt{m-1}}. \tag{5}$$

The function `codeTest2` keeps track of the elapsed time as it repeatedly encodes and decodes. Every five minutes, it approximately computes the relative standard error of the BER estimate. It stops if the relative standard error is < 0.1. It also stops if the BER estimate is at least two standard errors below 10^{-5}, indicating that the upper limit of a 95% confidence interval is $< 10^{-5}$.

Equation (5) cannot be used when $m = 0$. In this case, we estimate the BLER to be $(0.95)^{1/n}$ because this is the upper limit with 95% confidence. The function `codeTest2` conservatively estimates the

BEREB to be $1/2$,[9] and stops if the product of these two estimates is $< 10^{-5}$. None of our tests stopped with zero block errors. A more complicated formula should be used when $m = 1$, but we did not use such a formula; we used equation 5 for all $m > 0$. When $m = 1$ this formula evaluates to NaN (not a number) so it was not possible for a test to stop with $m = 1$.

The function `codeTest2` also saves the results to a MAT-file every 5 min. This is useful because if the execution is interrupted, at most, 5 min of data is lost. The function `codeTest2` can also reload the MAT-file and continue the test from the point it was last saved.

10.3 RESULTS OF CODETEST2

The blue lines in Figure 19 are waterfall plots for polar codes of rate $1/2$. The codes were constructed using AlgorithmD.java, and the BERs were estimated using `codeTest2`.[10] A blue line that meets the bottom of the graph indicates that a BER estimate less than 10^{-5} was computed.

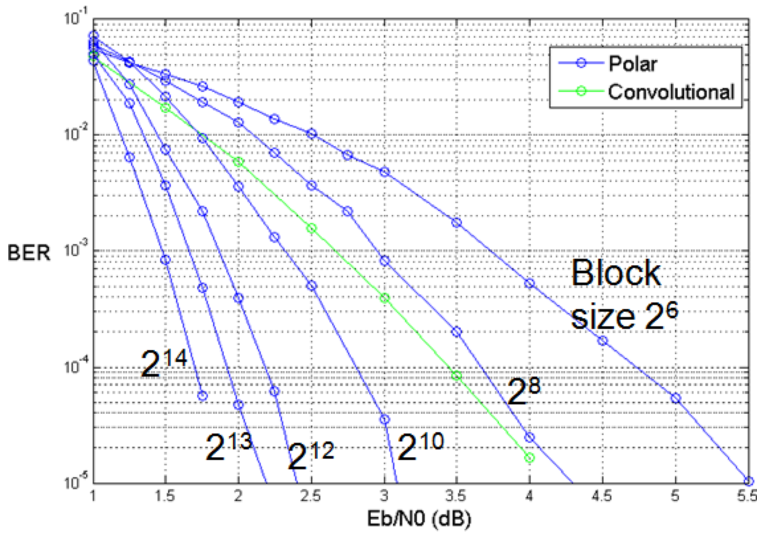

Figure 19. BER estimates of rate $1/2$ codes in Gaussian channels.

We mentioned previously that each line on a waterfall plot shows the results of the same code at different E_b/N_0 levels. However, in the blue lines of Figure 19, each point on the line represents a code constructed for that particular E_b/N_0. This is a common practice in the polar codes literature because polar codes are easily customized for different E_b/N_0 levels.

10.4 QUATERNARY PHASE SHIFT KEYING SIMULATIONS

We now give another example of a channel:

Example 3: Gaussian Channel with Quaternary Phase Shift Keying (QPSK). This channel represents QPSK modulation and a line-of-sight RF channel with negligible multipath. The QPSK modulator

[9]Getting a better estimate of BEREB is possible by extrapolating from other tests that produced more block errors. The function `codeTest2` can use a user-supplied BEREB bound instead of $1/2$. We did not use this feature.

[10]The green line is a waterfall plot for a convolutional code, which is explained in Section 10.4. It was computed using runQPSK.m.

first maps a pair of bits to a point in the in-phase/quadrature (I/Q) plane. There are several QPSK mappings. We use $(0,0) \mapsto (-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$, $(0,1) \mapsto (-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, $(1,0) \mapsto (\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}})$, $(1,1) \mapsto (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$. The I/Q point is then converted to a sinusoidal waveform that is transmitted. The receiver rescales the waveform to compensate for the attenuation. The demodulator maps the waveform back to I/Q space, resulting in $(\pm\frac{1}{\sqrt{2}} + n_I, \pm\frac{1}{\sqrt{2}} + n_Q)$, where n_I and n_Q are random noise variables that are assumed to be independent, normally distributed, with 0 mean and equal variance.

For any $\sigma > 0$, the *Gaussian QPSK channel* with noise variance σ^2 has input set $\mathcal{X} = \{0,1\}^2$, output set $\mathcal{Y} = \mathbb{R}^2$, and transition probabilities

$$W(y_1, y_2|u_1, u_2) = \frac{1}{\sigma^2 2\pi} \exp\left(-\frac{\left(y_1 + \frac{(-1)^{u_1}}{\sqrt{2}}\right)^2 + \left(y_2 + \frac{(-1)^{u_2}}{\sqrt{2}}\right)^2}{2\sigma^2}\right).$$

Note that this is not a binary channel because it has more than two inputs. It is memoryless, and symmetric in an appropriate sense.

Like the BPSK Gaussian channel, the QPSK Gaussian channel has the same $E_s/N_0 = \frac{1}{2\sigma^2}$. However, twice as much information is sent in each use of the channel, so $E_b/N_0 = \frac{N}{4\sigma^2 K}$.

Section 4.8.4 of [27] shows that each use of the QPSK Gaussian channel is equivalent to two consecutive uses of the BPSK Gaussian channel with the same E_b/N_0. So `codeTest2` can simulate QPSK if we interpret two consecutive bits as the I and Q components of the same signal. We also wrote runQPSK.m, a MATLAB® script that explicitly simulates QPSK. This script can simulate polar coding, and it can also simulate convolutional coding, using functions from the MATLAB® Communications Toolbox. An (N, K, l) *convolutional code* is an FEC code that takes K bits as input and outputs N bits. However, these N bits are determined not only by the input but also by the previous $l - 1$ inputs. Convolutional codes are not block codes.

Figure 20 shows BER estimates computed with runQPSK.m. These include polar codes of three different lengths. They also include two convolutional codes, a $(2, 1, 7)$ and a $(2, 1, 9)$. The former is used in several U.S. military communications systems (for example, see [26]).

10.5 SIMULINK® RESULTS

We built Simulink® models representing QPSK Gaussian channels. Figure 21 shows results for three different FEC codes: a $(1024, 512)$ turbo code, a $(64800, 32400)$ LDPC code, and a $(61, 51)$ Reed–Solomon code.

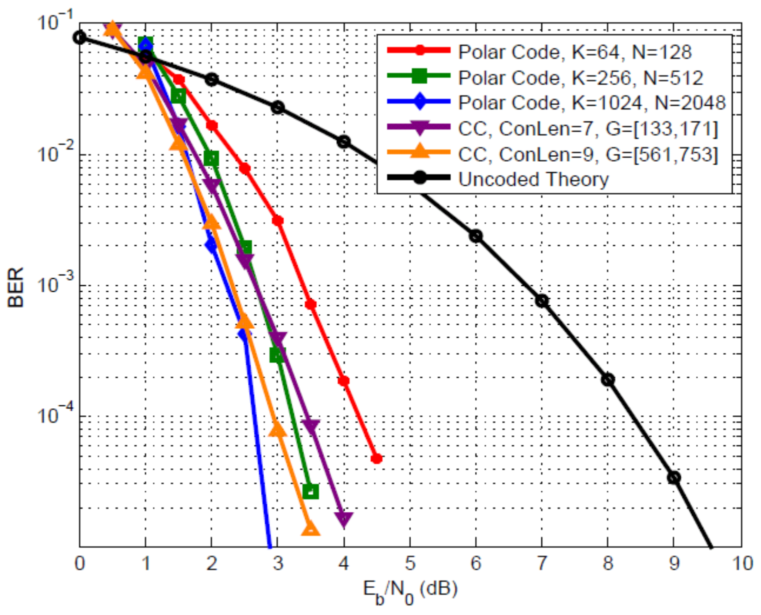

Figure 20. BER estimates of rate $1/2$ codes in QPSK Gaussian channels.

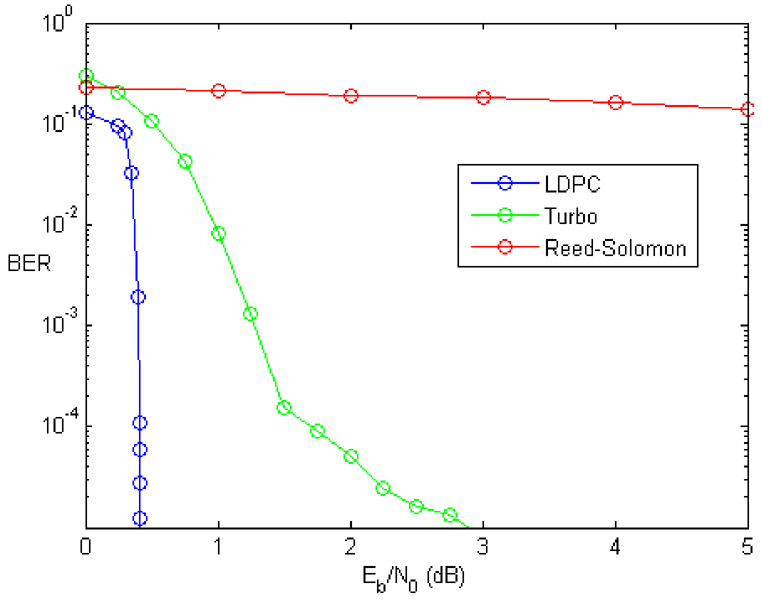

Figure 21. BER estimates in QPSK Gaussian channels.

11. CONCLUSION

Our FY14 progress has consisted mostly of implementing published algorithms and reproducing published results. We have implemented and tested several methods for encoding, decoding, and code construction.

Our preliminary conclusion is that polar codes can outperform the FEC currently used in some modes of Navy communications systems. Polar codes may also be able to outperform other codes that could be used to replace the current codes. Polar codes are more likely to be useful at higher data rates. Further research is needed to test polar codes, using more realistic models of Navy systems and their RF environments. We are now well positioned to do this further research.

REFERENCES

1. E. Arikan. 2009. “Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels,” *IEEE Transactions on Information Theory 55 (7)*:3051–3073.

2. D. Wasserman and A. Ahmed. 2014. “Polar Codes Source Codes.” Technical Document 3287. Space and Naval Warfare Systems Center Pacific (SSC Pacific), San Diego, CA.

3. R. W. Hamming. 1950. “Error Detecting and Error Correcting Codes,” *Bell System Technical Journal 29(2)*:147–160.

4. C. Shannon. 1948. “A Mathematical Theory of Communication,” *Bell Systems Technical Journal 27*:379–423, 623–657.

5. I. Tal and A. Vardy. 2011. “List Decoding of Polar Codes.” *Proceedings of the 2011 IEEE International Symposium on Information Theory (ISIT)* (pp. 1–5), July 31–August 5, St. Petersburg, Russia. IEEE.

6. P. Trifonov. 2012. “Efficient Design and Decoding of Polar Codes,” *IEEE Transactions on Communications 60(11)*:3221–3227.

7. A. Eslami and H. Pishro-Nik. 2011. “A Practical Approach to Polar Codes.” *Proceedings of the 2011 IEEE International Symposium on Information Theory (ISIT)* (pp. 16–20). July 31–August 5, St. Petersburg, Russia. IEEE.

8. K. Niu. and K. Chen. 2012. “CRC-Aided Decoding of Polar Codes,” *IEEE Communications Letters 16(10)*:1668–1671.

9. K. Niu, K. Chen, and J.-R. Lin. 2013 “Beyond Turbo Codes: Rate-Compatible Punctured Polar Codes.” *Proceedings of 2013 IEEE International Conference on Communications (ICC)* (pp. 3423–3427). June 9–13, Budapest, Hungary. IEEE.

10. C. Leroux, I. Tal, A. Vardy, and W. Gross. 2011. “Hardware Architectures for Successive Cancellation Decoding of Polar Codes,” *Proceedings of the 2011 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 1665–1668). May 22–27, Prague, Czech Republic. IEEE.

11. A. Alamdar-Yazdi and F. R. Kschischang. 2011. “A Simplified Successive-Cancellation Decoder for Polar Codes,” *IEEE Communications Letters 15(12)*:1378–1380.

12. C. Leroux, A. J. Raymond, G. Sarkis, I. Tal, A. Vardy, and W. J. Gross. 2012. “Hardware Implementation of Successive-Cancellation Decoders for Polar Codes,” *Journal of Signal Processing Systems 69 (3)*:305–315.

13. B. Yuan and K. Parhi. 2014. “Low-Latency Successive-Cancellation Polar Decoder Architectures Using 2-Bit Decoding,” *IEEE Transactions on Circuits and Systems I: Regular Papers 61(4)*:1241–1254.

17. Mishra, A., Raymond, A., Amaru, L., Sarkis, G., Leroux, C., Meinerzhagen, P., Burg, A., and Gross, W. 2012. “A Successive Cancellation Decoder ASIC for a 1024-Bit Polar Code in 180nm CMOS,” *2012 IEEE Asian Solid State Circuits Conference (A-SSCC)* (pp. 205–208).

18. Bravo-Santos, A. 2013. “Polar Codes for Gaussian Degraded Relay Channels,” *IEEE Communications Letters*, vol. 17, no. 2, pp. 365–368.

19. Tal, I. and Vardy, A. 2013. “How to Construct Polar Codes,” *IEEE Transactions on Information Theory*, vol. 59, no. 10, pp. 6562–6582.

20. Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduction to Algorithms*, vol. 2, MIT Press Cambridge.

21. Li, H. and Yuan, J. 2013. “A Practical Construction Method for Polar Codes in AWGN Channels,” *2013 IEEE TENCON Spring Conference* (pp. 223–226).

22. Wu, D., Li, Y., and Sun, Y. 2014. “Construction and Block Error Rate Analysis of Polar Codes Over AWGN Channel Based on Gaussian Approximation,” *IEEE Communications Letters*, vol. 18, no. 7, pp. 1099–1102.

23. Arikan, E. 2011. “Systematic Polar Coding,” *IEEE Communications Letters*, vol. 15, no. 8, pp. 860–862.

24. Sarkis, G. and Gross, W. 2013. “Increasing the Throughput of Polar Decoders,” *IEEE Communications Letters*, vol. 17, no. 4, pp. 725–728.

25. Guimaraes, D. A. 2010. *Digital Transmission: A Simulation-Aided Introduction with VisSim/Comm*, Springer.

26. Department of Defense (DoD). 2011. *Department of Defense Interface Standard MIL-STD-188-110C: Interoperability and Performance Standards for Data Modems, 23 September 2011.*

27. Sklar, B. 2001. *Digital Communications*, vol. 2, Prentice Hall NJ.

Dec

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-01-0188*

1. REPORT DATE *(DD-MM-YYYY)* December 2014	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)*
4. TITLE AND SUBTITLE Polar Codes		**5a. CONTRACT NUMBER**
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHORS David Wasserman		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) SSC Pacific, 53560 Hull Street, San Diego, CA 92152–5001		**8. PERFORMING ORGANIZATION REPORT NUMBER** TR 2054
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Innovative Science and Engineering (NISE) Program (Applied Research) SSC Pacific, 53560 Hull Street, San Diego, CA 92152–5001		**10. SPONSOR/MONITOR'S ACRONYM(S)**
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release.

13. SUPPLEMENTARY NOTES
This is work of the United States Government and therefore is not copyrighted. This work may be copied and disseminated without restriction.

14. ABSTRACT
The purpose of the project is to determine if polar codes can outperform the forward error correction (FEC) currently used in Navy wireless communication systems. This report explains polar codes to non-specialists.

The project team has written and tested software that implements several published polar coding algorithms. For comparison, the team has also implemented and tested other forward error correction methods: a turbo code, a low density parity check (LDPC) code, a Reed–Solomon code, and three convolutional codes.

The team's preliminary conclusion is that polar codes can outperform the FEC currently used in some modes of Navy communications systems. Polar codes may also be able to outperform other codes that could be used to replace the current codes. Polar codes are more likely to be useful at higher data rates. Further research is needed to test polar codes, using more realistic models of Navy systems and their RF environments. Space and Naval Warfare Systems Center Pacific is now well positioned to do this further research.

15. SUBJECT TERMS
Mission Area: Communications

polar codes	polar encoder	polar construction	Gaussian approximation method	polar coding algorithms
block codes	polar decoder	Tal/Vardy Method	forward error correction	

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** David Wasserman
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19B. TELEPHONE NUMBER** *(Include area code)*
U	U	U	U	53	(619) 553-3003

INITIAL DISTRIBUTION

84300	Library	(2)
85300	Archive/Stock	(1)
56270	D. Wasserman	(1)

Defense Technical Information Center Fort Belvoir, VA 22060–6218	(1)